Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jure Šemrov

Priprava programov OpenCL za učinkovito izvajanje na različnih arhitekturah

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič

Ljubljana, 2017

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

OpenCL postaja standardno okolje za hkratno delo z zelo različnimi napravami kot so procesorji, grafične procesne enote, različni pospeševalniki in celo programirljiva logična vezja. Izberite nekaj tipičnih aplikacij in nekaj različnih naprav ter za njih prilagodite programe OpenCL tako, da se bodo izvajali kar najbolj učinkovito. Na podlagi poskusov predlagajte rešitve, ki bi programerjem pomagale pri pisanju učinkovitih programov za najširši nabor strojne opreme.

Izjava o avtorstvu zaključnega dela

Spodaj podpisani Jure Šemrov, vpisna številka 63120073, avtor zaključnega dela z naslovom:

Priprava programov OpenCL za učinkovito izvajanje na različnih arhitekturah

IZJAVLJAM

1. da sem pisno zaključno delo študija izdelal samostojno pod mentorstvom izr. prof. dr. Uroša Lotriča;
2. da je tiskana oblika pisnega zaključnega dela študija istovetna elektronski obliki pisnega zaključnega dela študija;
3. da sem pridobil/-a vsa potrebna dovoljenja za uporabo podatkov in avtorskih del v pisnem zaključnem delu študija in jih v pisnem zaključnem delu študija jasno označil/-a;
4. da sem pri pripravi pisnega zaključnega dela študija ravnal/-a v skladu z etičnimi načeli in, kjer je to potrebno, za raziskavo pridobil/-a soglasje etične komisije;
5. soglašam, da se elektronska oblika pisnega zaključnega dela študija uporabi za preverjanje podobnosti vsebine z drugimi deli s programsko opremo za preverjanje podobnosti vsebine, ki je povezana s študijskim informacijskim sistemom članice;
6. da na UL neodplačno, neizključno, prostorsko in časovno neomejeno prenašam pravico shranitve avtorskega dela v elektronski obliki, pravico reproduciranja ter pravico dajanja pisnega zaključnega dela študija na voljo javnosti na svetovnem spletu preko Repozitorija UL;
7. dovoljujem objavo svojih osebnih podatkov, ki so navedeni v pisnem zaključnem delu študija in tej izjavi, skupaj z objavo pisnega zaključnega dela študija.

V Ljubljani, dne 5. januar 2017 Podpis študenta/-ke:

Iskreno se zahvaljujem mentorju izr. prof. dr. Urošu Lotriču za vso podporo, pomoč in čas, ki mi ga je posvetil pri izdelavi diplomske naloge. Posebna zahvala gre tudi staršem in bratu za vso podporo tekom študija.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPU	central processing unit	centralno procesna enota
GPU	graphics processing unit	grafično procesna enota
ALU	arithmetic logic unit	aritmetično logična enota
API	applications programming interface	programski vmesnik
SIMD	single instruction, multiple data	en ukaz, več podatkov
SIMT	single instruction, multiple thread	en ukaz, več niti
MIC	many integrated core	mnogojedrnik
ODI	on die interconnect	povezava med jedri
AVX	advanced vector extensions	napredna vektorska enota

Povzetek

V diplomski nalogi se posvečamo predvsem vprašanju, kako programe OpenCL napisati, da se bodo učinkovito izvajali na različnih arhitekturah. Težava, s katero se soočamo, so arhitekturne razlike med sistemi. Če torej želimo doseči maksimalno učinkovitost, moramo program ustrezno prilagoditi. Prilagoditve obsegajo število računskih enot, število niti v skupini, uporabo vektorske enote, lokalnega pomnilnika in predpomnilnikov ter še druge načine za prikrivanje latence. Na kratko, izkoristiti moramo morebitne arhitekturne prednosti naprave in paralelizem tako na nivoju ukazov, kot tudi na nivoju niti.

V nalogi obravnavamo pet programov, to so histogram, množenje matrik, predponska vsota, problem n teles in bitonično urejanje. Te programe prilagodimo trem različnim sistemom, in sicer CPE Intel Core i5-2450M, mnogojedrnik Xeon Phi 5110P, GPE Nvidia Tesla K20. Da bi te prilagoditve izkusili tudi v praksi, smo izmerili čas izvajanja programov za različno velike skupine in skušali razbrati kaj se dogaja.

Če naše ugotovitve posplošimo, lahko privzamemo, da naj bo število skupin vsaj toliko, kot je računskih enot, skupine pa naj bodo ravno prav velike, da zmanjšamo režijo preklopa skupin in pomnilniško latenco ter obenem ne povečamo režije zaradi komunikacije ali zmanjšamo števila skupin, ki se sočasno izvajajo na računski enoti. Za učinkovito izvajanje moramo na CPE in mnogojedrniku upoštevati predpomnilnike in širino vektorske enote, medtem ko moramo na GPE čim bolje izkoristiti visoko prepustnost ter prikriti latenco z velikim številom niti in lokalnim pomnilnikom.

Ključne besede: OpenCL, heterogeni sistemi, računske enote, delovne skupine, niti, SIMD, SIMT, lokalni pomnilnik.

Abstract

The main question in this thesis we will be trying to solve, is how to write a proper OpenCL program to effectively run on different architectures. A problem to overcome are the architectural differences between systems. To maximize the efficiency, we need to adapt the program. This defers by the number of compute units, number of threads in a work-group, use of a vector unit, local memory and cache to minimize latency. To summarize, we need to exploit both instruction and thread level parallelism as well as other architectural advantages.

We used five programs, histogram, matrix multiply, prefix sum, n body problem and bitonic sort. Then we adapted them to three different systems, Intel Core i5-2450M CPU, Xeon Phi 5110P manycore processor and Tesla K20 GPU. To test these adaptations in practice, we measured program runtime for different work-group sizes and tried to explain what is going on.

Our conclusions show, that we need at least as many work-groups as there are compute units. The work-group size have to be large enough to reduce the overhead of maintaining a work-group and hide memory latency. At the same time they should be small enough to reduce overhead of communication and to keep executing more work-groups simultaneously on each compute unit. To execute programs efficiently on a CPU and manycore processors, we need to take into account caches and wideness of a vector unit, while on a GPU we need to exploit high memory throughput and hide latency with large work-groups and local memory.

Keywords: OpenCL, heterogeneous systems, compute unit, work groups, work-items, SIMD, SIMT, local memory.

Poglavje 1

Uvod

Današnji računalniki poleg večjedrnih procesorjev vsebujejo še razne procesorje, specializirane za hitro digitalno procesiranje in grafično procesne enote, ki so bile v osnovi namenjene predvsem za hitro procesiranje kompleksih grafičnih operacij. Zmogljive grafične enote niso uporabne zgolj za zahtevne računalniške igrice, temveč tudi v znanosti. Tam se srečujemo z izvajanjem različnih zahtevnih računskih operacij, kot so modeliranje napovedi vremena in drugih fizikalnih problemov, procesiranje slik in videa ter računsko zahtevne matematične operacije v numeričnih metodah. Prvo podjetje, ki je izkoristilo potencial grafičnih enot, je bilo NVIDIA. S svojo izrazito paralelno usmerjeno platformo in izvajalnim okoljem CUDA je omogočilo izjemno pohitritev problemov, ki imajo visoko stopnjo podatkovnega paralelizma. Glavna težava platforme CUDA je, da ni odprtokodna in da je vezana na strojno opremo NVIDIA, torej je za druge proizvajalce in arhitekture neuporabna. Zato je združenje podjetij Apple, AMD, Intel, NVIDIA in drugih ustanovilo tehnološki konzorcij Khronos Group, ki je ustvarilo nov odprt standard in ogrodje za pisanje programov OpenCL, osnovano prav na arhitekturi GPE. Glavni izziv standarda je bil ustvariti izvajalno okolje in programski vmesnik, ki je dovolj posplošen, da omogoča izvajanje na širokem spektru računalniških arhitektur, kot sta recimo CPE in GPE. Hkrati mora tudi podpirati široko stopnjo paralelizma tako na nivoju niti kot ukazov ter se učinkovito izva-

jati tako na homogenih kot heterogenih sistemih. Besedna zveza heterogeni sistemi se nanaša prav na uporabo več različnih vrst procesorjev, da bi pridobili na boljši učinkovitosti izvajanja, kot tudi boljši energetski učinkovitosti. Tipičen heterogen sistem tako lahko predstavlja kombinacija CPE in GPE, pri čemer je CPE namenjena bolj splošnonamenskim zadevam, GPE pa je specializirana za izvajanje matematičnih operacij z visoko stopnjo podatkovnega paralelizma kot je grafika.

V nadaljevanju diplomske naloge bomo najprej opisali abstraktno arhitekturo OpenCL, na abstrakten način predstavili izvajanje programov OpenCL in definirali osnovne pojme. Tako bralca seznanimo z osnovnimi koncepti, ki so nujno potrebni za razumevanje diplomskega dela. Nato sledi opis načinov s katerimi lahko izkoristimo specifične enote strojne opreme in optimizacije v OpenCL. V tem poglavju najprej definiramo arhitekturne razlike med običajnimi CPE in GPE ter mnogojedrniki, kot je Xeon Phi. Te razlike so podlaga za prilagajanje programov posameznim arhitekturam, če želimo, da se program izvaja učinkovito. Prilagoditve obsegajo načine za skrivanje pomnilniške latence, uporabo lokalnega pomnilnika in vektorskih enot in podobno. Opišemo tudi pomen delovnih skupin in število niti v skupini ter kakšen vpliv ima to na računske enote in prikrivanje pomnilniške latence. To je zelo pomembno, če želimo doseči učinkovito izvajanje programa. Nato sledi predstavitev testnih primerov, v katerem bralca seznanimo s problemi, ki jih obravnavamo v praktičnem delu diplomske naloge, predstavimo pa tudi osnovno idejo za paralelizacijo problema. Sledi še praktičen del, kjer najprej opišemo tri arhitekture, za katere bomo na podlagi njihovih značilnosti optimizirali algoritme tako, da se bodo čim bolj učinkovito izvajali. Obenem bomo merili čas izvajanja pri različnem številu niti na skupino in te rezultate meritev komentirali ter poskušali razbrati, kaj se dogaja.

Poglavje 2

Ogrodje OpenCL

Glavni cilj standarda OpenCL je ustvariti platformo, ki vsebuje izvajalno okolje (ang. runtime environment) in programerska orodja tako, da bo dobro izkoristitil heterogenost in večjedrnost današnjih arhitektur. Poleg tega morajo biti programi prenosljivi med različnimi proizvajalci in napravami, izvajalno okolje pa mora imeti čim manjšo režijo.

Za dobro prenosljivost med sistemi OpenCL definira 4 modele: [3]

1. Platformni model: Model določa, da imamo en procesor namenjen koordinaciji izvajanja (gostitelj) ter vsaj en procesor, ki izvaja ščepec (naprava). Definira abstraktni model računalnika, ki ga uporablja programer za pisanje ščepcev.

2. Izvajalni model: Definira nastavitve izvajalnega okolja in način, kako se ščepci istočasno izvajajo na napravah. To vključuje kontekst na gostitelju za nadzor komunikacije med gostiteljem in napravami.

3. Pomnilniški model: Definira abstraktno pomnilniško hierarhijo, ki jo uporablja ščepec, neodvisno od dejanske pomnilniške hierarhije. Ta abstraktna pomnilniška hierarhija je še najbližja hierarhiji, ki jo imajo sodobne GPE.

4. Programski model: Določa, kako se sočasno izvajanje ščepca preslika na dejansko strojno opremo.

Gostitelj je ponavadi CPE, GPE pa predstavlja napravo, na kateri se paralelno izvaja ščepec. Platformni model definira prav to povezavo med gostiteljem in napravo ter predstavi napravo na abstrakten način. Gostitelj pripravi kontekst, ukazne vrste in ščepec za izvajanje na GPE in pri tem še določi stopnjo paralelizma. S tem je določen izvajalni model. Preko pomnilniškega modela se določi, kam v abstraktno pomnilniško hierahijo se rezervira prostor za posamezne podatke. Izvajalno okolje bo nato samodejno preslikalo posamezen abstrakten pomnilniški naslovni prostor na fizično napravo. Programerju tako ni treba skrbeti, kakšen tip naprave uporablja. Izvajalno okolje namreč samo določi, kako se bo sočasno izvajanje ščepca preslikalo na dejansko strojno opremo.

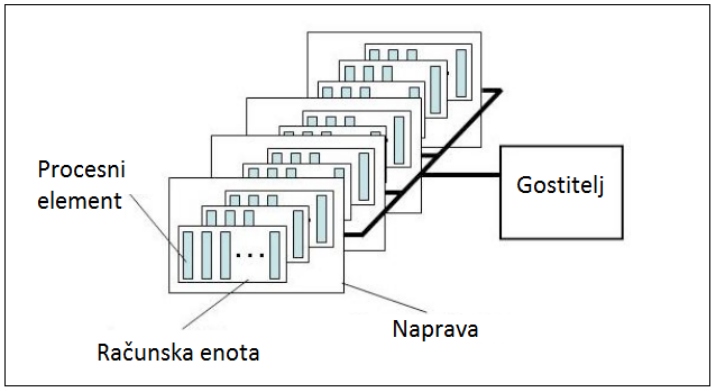
Vendar ima vsa ta abstrakcija, ki omogoča visoko prenosljivost med različnimi arhitekturami svojo ceno. Večja posplošenost modela pomeni, da imamo na voljo manj optimizacij, ki so na voljo zgolj na specifičnih napravah. Poleg tega se moramo zavedati, da je prenosljivost programov omejena. Program tako sicer lahko izvajamo na različnih arhitekturah, učinkovitost in hitrost izvajanja programa pa nista zagotovljeni. Tudi v splošnem velja, da sta prenosljivost in hitrost izvajanja kompromis, ki ga moramo sprejeti glede na to, kaj se nam zdi bolj pomembno.

V diplomskem delu, želimo ugotoviti, katere podatke o platformi uporabiti in kako, da bo koda tekla čim bolj učinkovito. Med drugim bomo poskušali izkoristiti posebne arhitekturne lastnosti, kot sta lokalni pomnilnik in vektorska enota.

2.1 Platformni model OpenCL

Platformo si lahko predstavljamo kot implementacijo programskega vmesnika OpenCL (API), specifično za določenega proizvajalca - platforma lahko torej upravlja samo z napravami istega proizvajalca kot je platforma sama. Seveda pa OpenCL podpira več, morda različnih naprav, združenih pod isto platformo.

Platformni model definira vlogo gostitelja, ki služi koordinaciji izvajanja in naprav, ki jih predstavi na abstrakten način. Vsako napravo si lahko predstavljamo kot tabelo med seboj neodvisnih računskih enot (ang. compute unit), ki so sestavljene iz več procesnih elemetov (ang. processing elements). Abstraktna predstavitev arhitekture OpenCL torej ustreza dejanski arhitekturi GPE. Za preslikavo na dejansko strojno opremo pa je odgovorno izvajalno okolje, ki ga specificira proizvajalec naprave.



Slika 2.1: Platformni model definira abstraktno hierarhijo naprav.

2.2 Izvajalni model OpenCL

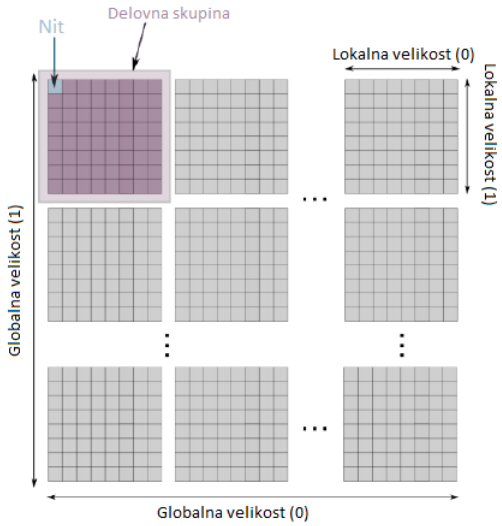
OpenCL program je sestavljen iz dveh delov, in sicer iz izvajalnega okolja, ki se izvaja na gostitelju, in ščepca, ki se izvaja na napravi.

Preden lahko gostitelj odda napravi zahtevo za izvedbo ščepca, mora definirati kontekst. Kontekst pripravi okolje za izvajanje ščepca in tako predstavlja nekakšno podatkovno strukturo za komunikacijo med gostiteljem in napravo. Izvajalni model vključuje množico naprav za izvajanje ščepca in mehanizem za njihovo koordinacijo, upravlja pomnilniškime objekte in preko ukaznih vrst izvaja operacije. Vsaka naprava ima vsaj eno ukazno vrsto, ki služi koordinaciji izvajanja ščepcev na napravi. Preko ukazne vrste se na napravo pošiljajo ukazi gostitelja, kot so zagon ščepca na napravi, ukazi za

prenos podatkov med gostiteljem in napravo. Določi se tip sinhronizacije, ki določa vrstni red izvajanja ukazov.

V izvajalnem modelu definiramo način izvajanja ščepeca na napravi in organizacijo niti. Bistvo ščepca je namreč, da se izvede nad določenim številom niti (ang. work-item), kjer vsaka nit reši majhen del problema. Vsaka nit ima svoj enolični globalni indeks, s katerim se identificira. Glede na problem določimo, kakšna bo dimenzija in vsaki dimenziji določimo velikost. OpenCL podpira eno, dve ali tri dimenzije. Dimenzijo določimo tako, da čim bolje preslikamo realni problem na izvajanje ščepca. Ena dimenzija tako recimo predstavlja tabelo števil, dve dimenziji pa predstavljata 2D matriko kot rezultat pri množenju matrik.

Posamezne niti organiziramo po delovnih skupinah (ang. work-group) tako, da ima vsaka skupina enako število niti. Vsaka delovna skupina ima svoj enolični indeks, prav tako ima vsaka nit znotraj iste skupine še svoj enolični lokalni indeks. Skupaj tako enolično določata globalni indeks niti. Velikost skupine je pomemben podatek, saj ima lahko velik vpliv na razvrščanje niti na računske enote. Delovna skupina se namreč izvede na eni računski enoti, medtem ko se lahko več skupin hkrati izvaja na eni računski enoti. Izbrano število niti v bloku vpliva na izkoriščenost računske enote in s tem na učinkovitost izvajanja. Optimalno število niti je precej odvisno od arhitekture naprave, velik pomen pa ima tudi ščepec. Premajhno število niti v skupini pomeni, da ima pomnilniška latenca velik vpliv na čas izvajanja, saj računske enote niso dovolj dobro izkoriščene. Če je skupina prevelika, pa je problem večja poraba virov in komunikacija med nitmi. Pomembno dejstvo je tudi, da so posamezne skupine med seboj neodvisne. To pomeni, da se lahko več skupin vzporedno izvaja vsaka na svoji računski enoti. Iz tega sledi, da naj bo število skupin vsaj toliko, kolikor je računskih enot. S tem razvrščevalniku damo več prožnosti pri razvrščanju za ceno večje režije pri preklopu opravil.



Slika 2.2: Vizualna predstavitev izvajalnega modela OpenCL.

2.3 Pomnilniški model OpenCL

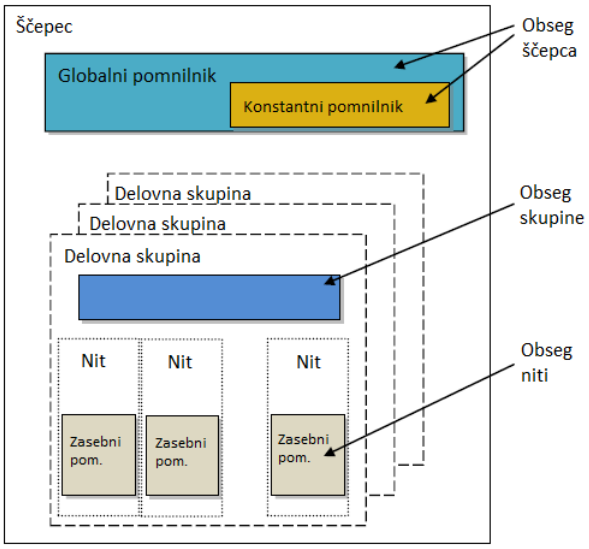
Ena glavnih lastnosti OpenCL je strojna neodvisnost in prenosljivost programov. To pomeni, da je bilo treba nekako rešiti problem arhitekturne razlike pomnilniških hierarhij med različnimi vrstami naprav. CPE se zanaša na strojno upravljanje s predpomnilnikom in je popolnoma koherenten sistem, GPE pa se bolj zanaša na eksplicitni predpomnilnik, ki ga upravlja programer. Da bi ohranil prenosljivost, OpenCL definira abstraktno pomnilniško hierarhijo, ki jo uporabljajo programerji med pisanjem ščepca. Na strani proizvajalca je, da to preslika na dejansko pomnilniško hierarhijo. Abstraktni model definira 4 različne tipe pomnilniškega prostora, ki določajo, kam se bo preslikal posamezen naslovni prostor.

Prvi v vrsti in največji, a hkrati tudi najpočasnejši v hierarhiji je globalni pomnilnik (ang. global memory). Njegova značilnost je, da je viden vsem enotam in da nima zagotovljene konsistentnosti med nitmi v različnih delovnih skupinah. Nit iz skupine A nima zagotovila, da bo videla pravilno stanje niti iz skupine B. Konsistentnost z globalnim pomnilnikom lahko zagotovimo samo znotraj skupine, in sicer z uporabo prepreke. Vsak prenos

podatkov iz/na gostitelja se opravi na globalnem pomnilniku. Del glavnega pomnilnika predstavlja konstantni pomnilnik (ang. constant memory) in je namenjen izključno bralnemu dostopu. Namenjen je hranjenju podatkov, do katerih hkrati dostopajo vse niti, kot so recimo konstantne spremenljivke.

Naslednji v hierarhiji je lokalni oziroma deljeni pomnilnik (ang. local memory). Je precej majhen v primerjavi z globalnim pomnilnikom, zato pa ima vsaka računska enota svoj lastni lokalni pomnilnik. Lokalni pomnilnik si delijo vse niti v delovni skupini in služi sinhronizaciji niti v isti skupini z uporabo preprek ter hranjenju skupnih podatkov. Lokalni pomnilnik je namenjen napravam, ki omogočajo uporabo eksplicitnega predpomnilnika, kot so GPE. Na teh napravah z uporabo lokalnega pomnilnika, ki ima precej manjšo latenco kot globalni pomnilnik, precej pospešimo hitrost programa. Arhitekture, ki ne poznajo take vrste predpomnilnikov, kot so CPE, tipično nimajo koristi iz uporabe lokalnega pomnilnik. Programe je treba napisati tako, da upoštevajo predpomnilniško hierarhijo, ki je v celoti nadzirana s strani CPE. Nasprotno, uporaba lokalnega pomnilnika v tem primeru zgolj poveča režijo zaradi kopiranja, zato v takem primeru uporabljamo lokalni pomnilnik zgolj za komunikacijo med nitmi v skupini.

Zasebni pomnilnik (ang. private memory) je še zadnji in najmanjši v pomnilniški hierarhiji in pripada samo eni niti. Privzeto so to vse lokalne spremenljivke in argumenti ščepca. Kam se dejansko preslika, je odvisno od naprave. Na CPE je zaradi majhnega števila registrov del sklada procesa (ang. stack), na GPE pa je to del posebnega pomnilnika, sestavljenega iz velikega števila registrov.



Slika 2.3: Abstraktna predstavitev pomnilniškega modela OpenCL

2.4 Programski model OpenCL

Določa, kako se izvajanje ščepcev preslika na strojno opremo. To obsega predvsem dodelitev skupin računskim enotam in je odvisno od vrste naprave in proizvajalca.

2.5 OpenCL ščepec

Program, ki se izvaja na napravi, se imenuje ščepec. Posamezen ščepec ni pravzaprav nič drugega kot C-jevska funkcija z dodatnim predpono *__kernel*. Izvorna koda je podana kot niz znakov in je ponavadi ločena v posebni tekstovni datoteki zaradi boljše modularnosti. Ščepec se prevede v času izvajanja gostiteljskega programa (ang. JIT, just-in-time compilation). To omogoča, da prevajalnik optimizira ščepec za specifično napravo in s tem izkoristi posebne enote naprave, kot je recimo vektorska enota, poleg tega pa omogoča spreminjanje ščepca brez prevajanja osnovnega programa. Ker je prevajanje ščepca precej potratno, nam izvajalno okolje omogoča uporabo že preve-

denega ščepca v obliki binarne kode. Če bi želeli v tem primeru ohraniti prenosljivost in program izvajati na različnih platformah, moramo binarno kodo za posamezne platforme prenašati skupaj s programom.

Sam programski jezik za pisanje ščepcev je izpeljanka programskega jezika C99 z nekaterimi omejitvami kot so nepodprtost rekurziji, kazalcem na funkcije ipd.

Poglavje 3

Strojna oprema in ogrodje OpenCL

OpenCL je bil razvit z namenom pisanja paralelnih programov s poudarkom na prenosljivosti programov med različnimi arhitekturami. Vsaka arhitektura ima specifične lastnosti, ki imajo tako svoje prednosti kot slabosti. To v praksi pomeni, da se program, optimiziran za učinkovito izvajanje na GPU, morda ne bo učinkovito izvajal na CPU. Pri optimizaciji kode je treba upoštevati prednosti in omejitve arhitekture naprave, kot so recimo število procesnih enot (jeder), velikost glavnega pomnilnika, upoštevati morebitne predpomnilnike in vrste predpomnilnikov, pasovna širina glavnega vodila, kompleksnost procesnih enot ipd. V tem poglavju bomo naprej definirali osnovne arhitekturne razlike med CPE in GPE. Ravno zaradi teh razlik je pomembno, da program prilagodimo za izvajaje na posamezni arhitekturi. Prilagoditve obsegajo število niti v skupini, število skupin, načini dostopa do pomnilnika, upoštevanje pomnilniške hierarhije, kot je predpomnilnik in lokalni pomnilnik. Upoštevati moramo tudi sam algoritem. Če je algoritem v večji meri sestavljen iz V/I operacij, potem moramo čim bolje izkoristiti princip lokalnosti in dostope do pomnilnika čim bolj prilagoditi arhitekturi. Tako bolje izkoristimo pomnilniško prepustnost. Še ena stvar je uporaba vektorske enote, s katero lahko izkoristimo še paralelizem na nivoju ukazov,

če arhitektura podpira vektorske ukaze.

3.1 Bistvene arhitekturne razlike med CPE in GPE

CPE in GPE sta dve različni procesorski arhitekturi, pri čemer ima vsaka svoje cilje, ki jih želi doseči in posledično različne kompromise, ki jih morata pri tem sprejeti. Če je CPE v svoji osnovi namenjen hitremu in učinkovitemu serijskemu izvajanju programov in predstavlja precej kompleksno enoto, je GPE namenjen paralelnemu procesiranju tipa SIMT (ang. single instruction, multiple thread), kjer več procesnih enot hkrati izvaja isti ukaz na različnih podatkih. Čeprav je posamezno jedro na GPE precej bolj enostavno in počasnejše v primerjavi z jedrom CPE, ima GPE več sto ali celo več tisoč jeder, oz. procesnih enot, ki lahko sočasno izvajajo ukaze. V nadaljevanju bomo na splošno opisali arhitekturne razlike med CPE in GPE, prednosti in slabosti posameznih arhitektur in pri katerih opravilih se posamezna arhitektura izkaže za boljšo. Te razlike bistveno vplivajo na način izvajanja programov in jih moramo upoštevati pri pisanju OpenCL programov, če želimo, da se na dani platformi program izvaja čim bolj učinkovito.

CPE predstavlja splošnonamenski procesor, namenjen širokemu spektru opravil, od najbolj splošnega računanja do nadzora nad celotnim sistemom. Skupaj z operacijskim sistemom zagotavlja učinkovito uporabo virov. Današnje CPE so zelo kompleksne in dosegajo precej višje frekvence delovanja od GPE. Ker so bolj specializirana za sekvenčno izvajanje ukazov, imajo veliko dodatne logike, podpirati pa morajo kopico elementov, ki jih za delovanje potrebujejo operacijski sistemi. Med logiko za hitrejše procesiranje lahko vključimo paralelizem na nivoju ukazov z n-stopenjskim cevovodom, kjer se n ukazov izvaja hkrati na različnih stopnjah cevovoda. Seveda cevovod ni popolna rešitev, saj tu pogosto nastopijo cevovodne nevarnosti, ko več stopenj cevovoda uporablja isto enoto, podatkovnih ter kontrolnih nevarnosti, kjer moramo zaradi skokov izprazniti cevovod. Da bi se izognili tem nevar-

nostim, procesorji uporabljajo rešitve, kot so špekulativno izvajanje ukazov, dinamično razvrščanje ukazov in dinamično predikcijo skokov. Te rešitve omogočajo procesorju, da med izvajanjem spreminja vrstni red med seboj neodvisnih ukazov in glede na zgodovino skokov napove izid bodočih. Današnje CPE običajno ponujajo tudi paralelizem tipa SIMD (ang. single instruction multiple data), kjer z uporabo vektorskih enot izkoristimo podatkovni paralelizem. Ta CPE omogoča, da zgolj z enim ukazom izvede eno operacijo nad več podatki. Število podatkov, ki jih lahko naenkrat obdela, je omejeno s širino vektorske enote, tipično 128 ali 256 bitov. Uporaba SIMD ukazov lahko precej pospeši multimedijske operacije ipd. Večjo zmogljivost CPE dosegamo tudi z uporabo strojne večnitnosti, ki omogoča boljšo izkoriščenost procesorja, saj si lahko več niti deli funkcijske enote procesorja, kar je koristno v primeru čakanja ene niti. V tem primeru lahko razvrščevalnik, ki je implementiran kot del operacijskega sistema, enostavno preklopi na drugo nit.

Ne smemo pozabiti na upravljanje z večnivojsko pomnilniško arhitekturo. Podatki so namreč lahko shranjeni v glavnem pomnilniku, predpomnilniku, ali pa v najslabšem primeru na disku. CPE ima na voljo le majhno število računskih enot. Za svoje delovanje potrebuje veliko količino glavnega pomnilnika, pa tudi predpomnilnik ima precej velik pomen. Predpomnilnik je majhen in več nivojski, a zelo hiter pomnilnik, saj je čas dostopa do predpomnilnika bistveno krajši kot do glavnega pomnilnika. Upoštevanje predpomnilnika pri optimizaciji programov je pomembno, saj poleg dodatne logike za izvajanje ukazov učinkovito prikriva pomnilniško latenco, to je čas dostopa do glavnega pomnilnika. Upravljanje s celotno pomnilniško hierarhijo je samodejno in v celoti prepuščeno strojni opremi v CPE. Predpomnilnik na najnižjem nivoju je tipično razdeljen na ukazni in podatkovni del. Taka izvedba se imenuje Harvardska arhitektura in preprečuje ozko grlo, zaradi katerega pride pri velikem številu dostopov preko enega vodila. Novejše CPE, ki jih uporabljamo danes, so večinoma večjedrne, tipično imajo od 2 do 4 jedra. S tem paralelnost CPE doseže svoj pravi cilj, saj lahko vsako jedro

izvaja svojo nit vzporedno in neodvisno od drugih jeder. Glavni vzrok za razvoj paralelnih arhitektur je, da ne moremo več učinkovito povečevati frekvence procesorjev. Višja frekvenca je povezana z višjo porabo električne energije, kar vodi v pregrevanje. Posledično se danes frekvence procesorjev ne zvišujejo več, kar pa proizvajalci skušajo nadomesti z več jedri.

Če je CPE zasnovana za hitro sekvenčno računanje in s kompleksnimi strojnimi rešitvami za podporo operacijskim sistemom, je arhitektura GPE zasnovana za učinkovito paralelno izvajanje velikega števila niti. Sprva so bile grafične enote sestavljene iz velikega števila senčilnikov, točk in vozlišč, omejene zgolj hitro obdelovanje grafičnih operacij. Te so bile nato posredovane neposredno na zaslon in niso omogočale prenosa obdelanih podatkov nazaj na gostitelja. Z razvojem tehnologije pa smo dobili grafične enote, ki so omogočale tudi splošnonamensko računanje (ang. General-purpose computing on graphics processing units). Moderna GPE je sestavljena iz velikega števila procesnih elementov (ang. processing element) oz. jeder, organiziranih po računskih napravah (ang. compute unit), ki lahko tako sočasno izvajajo več sto niti. Posamezna računska naprava lahko izvaja več skupin, posamezna skupina pa je omejena na izvajanje na eni računski enoti. Organizacija niti je pomembna, saj z njo vplivamo na skalabilnost programa, to je, kako dobro bomo izkoristili računske enote in njihove procesne elemente ter kako dobro bomo prikrili pomnilniško latenco. Posamezen blok se na računski napravi razdeli na več snopov niti (ang. warps[1]), razvrščevalnik niti (ang. warp scheduler) pa skrbi za razvrščanje posameznih snopov za izvajanje. V primerjavi s CPE imajo grafične enote strojni razvrščevalnik niti. S tako implementacijo je preklapljanje med snopi niti precej hitrejše in bolj učinkovito, kar je pri velikem številu niti precej pomembno. Taka arhitektura ustreza SIMT (single instruction, multiple thread) modelu, en ukaz se torej preslika na več niti [11]. Arhitektura SIMT je pravzaprav vrsta SIMD, glavna razlika je v tem, da SIMT doseže večji izkoristek procesne enote z uporabo večnitnosti, SIMD pa z uporabo večje enote, ki lahko hkrati izvede

[1] AMD uporablja izraz “wavefront”.

eno operacijo nad več podatki. V SIMT arhitekturi vse niti v snopu izvajajo isti ukaz. To pomeni, da si niti delijo pomnilnik ter logiko za prevzem, dekodiranje in izvršitev ukaza, kar postane težava pri divergenci v programih. V tem primeru morajo niti v istem snopu, ki nimajo izpolnjenih pogojev počakati ostale niti, saj jih GPE dezaktivira. Poenostavljeno, to pomeni, da se morebitna divergenca v takem primeru ne izvede paralelno, ampak serijsko, kar lahko precej upočasni izvajanje. Dejanska implementacija GPE se seveda razlikuje od proizvajalca do proizvajalca. NVIDIA GPE uporabljajo izključno SIMT model izvajanja in so zato skalarne narave, medtem ko so AMD GPE vektorske narave in SIMT model izvajanja kombinirajo s SIMD modelom [17, 21, 22].

S stališča posameznih jeder GPE niso nič posebnega, saj so v primerjavi s CPE bolj enostavna in imajo precej manjšo frekvenco. Tako ne vključujejo dodatne logike, kot je špekulativno izvajanje in dinamično razvrščanje ukazov. Prav ta enostavnost omogoča, da je jeder veliko, a so GPE kljub temu s stališča porabe električne energije še vedno zelo učinkovite. Pomnilniški model je več nivojski, sestavljen iz globalnega pomnilnika, na vsaki računski enoti pa iz lastnega, majhnega in eksplicitnega predpomnilnika nivoja L1, na katerega lahko programer z uporabo lokalnega pomnilnika na zahtevo sam prenese podatke iz globalnega pomnilnika. Z uporabo lokalnega pomnilnika se programer izogne visoki latenci v primeru neporavnanih ali naključnih dostopov do globalnega pomnilnika. Modernejše GPE vsebujejo tudi strojno voden predpomnilnik nivoja L2, ki si ga delijo vse računske enote, vendar pa je v primerjavi s predpomnilnikom na CPE precej manjši. Pomnilniško latenco na GPE torej bolj kot z velikimi predpomnilniki prekrivamo z uporabo velikega števila niti in lokalnim pomnilnikom [14].

Za GPE je zelo pomembno, da ima glavno vodilo zelo veliko pasovno širino (ang. bandwidth). Veliko število niti, ki se hkrati izvajajo, predstavljajo velik pritisk za pomnilnik. Pomnilnik tako predstavlja ozko grlo na grafičnih enotah. Računsko zahteven problem na CPE se v tem primeru spremeni v pomnilniško zahteven problem na GPE, kar je pravzaprav glavna težava vseh

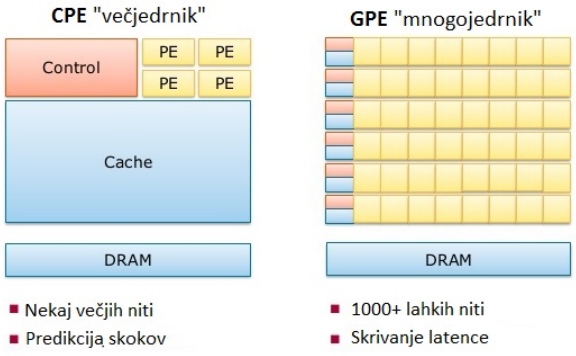
mnogojedrnikov.[2] Slaba prepustnost in visoka latenca povzročata stradanje procesnih enot, enote v tem primeru niso dovolj dobro izkoriščene [14]. Za primerjavo, latenca CPE do glavnega pomnilnika je nekaj 100 urinih ciklov, medtem, ko je latenca na GPE tipično reda nekje od 400 do 800 urinih ciklov.

Čeprav lahko GPE precej pohitrijo izvajanje programov, je treba poudariti, da so zelo ozko specializirane, zato niso rešitev za vse težave. So izrazito paralelno usmerjene in oblikovane predvsem za hitro računanje v problemih z visoko stopnjo podatkovnega paralelizma. Na drugi strani zelo slabo izkažejo v zaporednem svetu. GPE imajo v primerjavi s CPE še eno veliko prednost, in sicer glede kompatibilnosti za nazaj. Medtem ko morajo moderne CPE ohraniti kompatibilnost strojne opreme za desetletja nazaj, se lahko strojna oprema GPE spremeni praktično z vsako novejšo arhitekturo, kompatibilnost pa mora obdržati le na API nivoju. To je tudi glavni razlog za tako hiter razvoj GPE.

Slika 3.1 prikazuje glavne razlike med CPE in GPE. CPE ima majhno število procesnih enot za majhno število niti, kompleksno kontrolno enoto, latenco pa prikrivajo z večjimi in večnivojskimi predpomnilniki. GPE ima precej veliko število bolj enostavnih procesnih enot, razdeljenih po računskih enotah. Posamezne računske enote si delijo lokalni pomnilnik in enostavnejšo kontrolno enoto. Namenjene so sočasnemu izvajanju velikega števila niti z možnostjo hitrega preklapljanja, zato predpomnilniki nimajo velikega pomena. Niti na GPE so torej lahke, kar pomeni, da si delijo naslovni prostor in druge vire tako, da je režija pri zamenjavi konteksta čim manjša [14].

Obstaja še ena razlika z vidika registrov. S stališča CPE je register najdražji, najmanjši in najhitrejši element, kamor se shrani podatek za obdelavo. Tipično ima vsako jedro od 8 do 16 splošnonamenskih registrov, saj so zaradi izjemno nizke latence tudi precej dragi. Tu je hitrost najpomembnejši faktor, saj zaporedni ukazi pravzaprav komunicirajo preko registrov. Tipična GPE ima 16k do 32k registrov, optimiziranih za visoko prepustnost, ki pa

[2] "A many core processor ≡ A device for turning a compute bound problem into a memory bound problem" – Kathy Yelick, UC Berkeley



Slika 3.1: Primerjava CPE in GPE

imajo višjo latenco in so zato počasnejši, a cenejši. Na GPE višja latenca registrov ne predstavlja večjih težav, saj lahko latenco prekriva z menjavo snopa niti. Med dvema zaporednima ukazoma v istem snopu niti tako mine več urinih ciklov, zato počasnejši registri niso omejitev. Iz strojnega vidika so registri na GPE bolj neka vrsta pomnilnika kot registri, ki jih poznamo v CPE. Pravzaprav je kodiranje ukazov z registri edina podobnost med registri CPE in GPE. Z uporabo registrov omejimo število bitov za kodiranje naslova v ukazu [21].

3.2 Mnogojederni procesorji

Posebno vrsto sistemov predstavljajo mnogojederni procesorji. So nekakšna specializacija klasičnih večjedrnih CPE. Optimizirani so za paralelizem z uporabo velikega števila jeder in visoko pomnilniško prepustnost za ceno večje latence in precej manjše frekvence posameznih jeder. Tipično imajo mnogojederniki od 50 do 100 jeder z dobro podporo za sočasno večnitnost (ang. simultaneous multithreading), zato so precej dragi in so namenjeni predvsem za znanstvenoraziskovalna področja. Visoka latenca se torej rešuje z uporabo velikega števila niti in večjimi ter večnivojskimi predpomnilniki. Da bi taka arhitektura sploh lahko izkoristila tako veliko število jeder, mora imeti hitre povezave in dobro topologijo povezav med jedri. V nasprotnem primeru bi

pomnilnik predstavljal ozko grlo, jedra pa bi ostala slabo izkoriščena. Tudi mnogojedrnki izkoriščajo vektorske enote, le da so te tipično precej širše.

GPE so za razliko od mnogojedrnih CPE bolj ozko specializirana, saj so namenjena predvsem algoritmom z visoko stopnjo podatkovnega paralelizma, kot je grafika, posamezna jedra pa so bolj enostavna. Mnogojederne CPE bolje izkažejo se pri problemih, kot so operacije z drevesi, seznami ipd., pa tudi nasploh v zaporednem svetu.

3.3 Niti in delovne skupine

Velikost delovne skupine lahko precej vpliva na hitrost izvajanja programa, zato predstavlja pomemben del premisleka pri pisanju učinkovitih programov. Izbira prave velikosti delovne skupine je načeloma težek problem, pri čemer moramo upoštevati algoritem (ali je program bolj računsko zahteven, ali ima več V/I operacij) ter arhitekturo in pomnilniško hierarhijo naprave. Premajhno število niti ne more dovolj dobro prikriti pomnilniške latence in poveča učinek režije preklapljanja med skupinami. Po drugi strani preveliko število niti povzroči več režije pri sinhronizaciji niti v skupini, še posebej, kadar ščepec vsebuje prepreke. Prepreke morajo namreč izvesti lahko menjavo konteksta, kjer se shranijo stanja vseh niti v skupini. Dodatno prepreke sinhronizirajo dostope do pomnilnika, zato večje število niti pomeni več sinhronizacije. Z izbiro prave velikosti delovne skupine do določene mere prikrijemo latenco, saj ima razvrščevalnik na voljo več niti za izvajanje, prevajalniku pa omogočimo implicitno vektorizacijo kode. Sama velikost delovne skupine je navzgor omejena s strojnimi omejitvami naprave in omejitvami iz samega ščepca. Ščepec, ki je precej računsko zahteven, naj ima manj niti na skupino kot ščepec, ki je bolj zahteven z V/I operacijami. Niti v isti skupini si namreč delijo vire kot so registri, ALE enote in lokalni pomnilnik na računski enoti, zato lahko prevelika poraba registrov in lokalnega pomnilnika zreducira število delovnih skupin, ki se sočasno izvajajo na računski enoti [1].

Za posamezno delovno skupino je značilno, da se lahko izvaja na eni

računski enoti, medtem ko se lahko več skupin izvaja sočasno na isti računski enoti. Iz tega sklepamo, da je dobro imeti vsaj toliko skupin, kolikor imamo računskih enot, saj lahko le tako dovolj dobro izkoristimo napravo. Veliko število skupin ponuja več prilagodljivosti pri razvrščanju skupin na računske enote, za ceno večje režije med preklapljanjem skupin. Na napravah, ki imajo malo računskih enot torej velja premisliti o tem, kako bi zmanjšali število skupin, medtem ko na mnogojedrnikih to ne velja, saj bi s tem samo slabše izkoristili napravo. Ena izmed možnih poenostavitev je, da kot argument, ki določa velikost skupine, postavimo na $NULL$ in tako velikost skupine raje prepustimo izvajalnemu okolju. Izvajalno okolje v tem primeru samo izbere optimalno število niti. Načeloma ne obstaja idealen recept, ki bi zagotavljal optimalno število niti. Zato je treba s številom niti v skupini eksperimentirati, pri čemer se je pametno držati priporočil proizvajalca.

GPE sestavlja veliko število procesnih enot združenih po računskih enotah, saj so ustvarjene za izvajanje velikega števila niti. Na GPE se niti izvajajo po snopih (ang. warp size), zato je zaželeno, da je velikost delovne skupine večkratnik tega števila. To število je tipično 32 za NVIDIA GPE in 64 za AMD GPE [3]. Gre za nekakšen paralelizem tipa SIMD, torej se en ukaz izvede na celotnem snopu niti. Zaželeno je, da je teh snopov v skupini čim več, saj lahko le tako dobro prikrijemo visoko latenco GPE. V primeru čakanja lahko razvrščevalnik enostavno preklopi na drug snop. V splošnem velja, da naj bo število niti od 128 do 256 ali več. Veliko število delovnih skupin za GPE ni problem, saj s tem bolje izkoristimo posamezne računske enote, na katerih se lahko sočasno izvaja več delovnih skupin [15].

CPE ima tipično le nekaj jeder, so pa ta precej hitrejša in se za prikrivanje latence zanašajo predvsem na predpomnilnik. Za dobro učinkovitost skupin, se priporoča vsaj od 64 do 128 niti na skupino. Velikost skupine vpliva na uspešnost samodejne vektorizacije kode, zato standard priporoča, naj bo večkratnik števila 4 oziroma 8, odvisno od dolžine vektorske enote [1]. Zaradi majhnega števila jeder je včasih morda bolje zmanjšati število skupin tako, da posamezna nit opravi več dela. To ne velja za mnogojedrne procesorje,

saj moramo čim bolje izkoristi veliko število računskih enot.

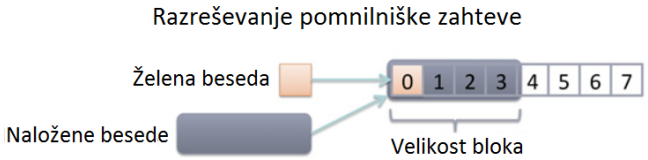
OpenCL omogoča poizvedbo s parametrom CL_KERNEL_PREFERRED _WORK_GROUP_SIZE_MULTIPLE v funkciji $clGetKernelWorkGroupInfo()$ priporočljivem večkratniku števila niti v skupini, ki ga lahko upoštevamo kot namig pri določanju števila niti na skupino.

3.4 Optimizacija pomnilniških dostopov

Pri razvoju učinkovitih programov, je poznavanje pomnilniške hierarhije in principov delovanja ključnega pomena. OpenCL zaradi svoje heterogenosti definira abstraktni pomnilniški model, sestavljen iz globalnega, lokalnega in zasebnega naslovnega prostora, za pravilno fizično preslikavo pa poskrbi sama implementacija OpenCL, ki jo specificira proizvajalec.

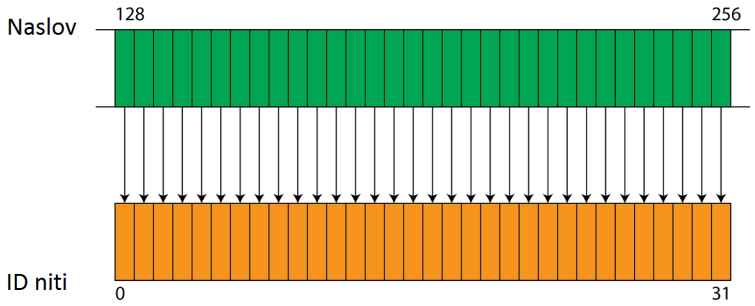
3.4.1 Globalni pomnilnik

Globalni pomnilnik predstavlja največji, a hkrati tudi najpočasnejši pomnilnik v hierarhiji, torej ustreza delovnemu pomnilniku na CPE in GPE. Glavna značilnost, ki jo moramo upoštevati, je, da se pomnilniške transakcije na obeh arhitekturah opravijo v blokih in ne po posameznih besedah. Razlog je ta, da je čas dostopa do globalnega pomnilnika precej visok v primerjavi s časom izvajanja posameznega ukaza, poleg tega pa so dostopi do pomnilnika statistično gledano precej zaporedni. S tem lahko precej zmanjšamo število dostopov do glavnega pomnilnika in prikrijemo latenco, vendar le, kadar so dostopi poravnani in ne preveč razpršeni.



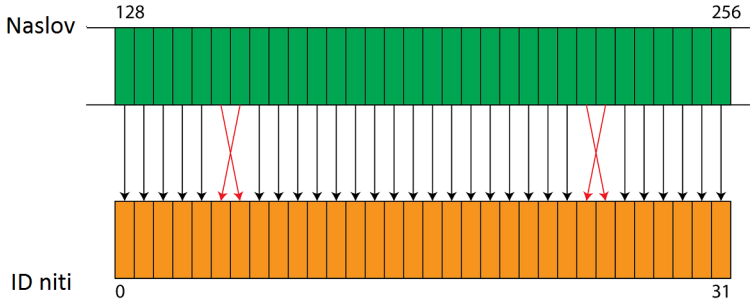
Slika 3.2: Pomnilniške transakcije se opravljajo v blokih

Upoštevanje visoke latence je izjemno pomembno na GPE, saj je tipično večja od latence CPE. GPE združujejo dostop (ang. memory coalescing) do globalnega pomnilnika z nitmi po snopih tako, da kar se da minimizirajo število transakcij. V praksi to pomeni, da bolj, kot bodo dostopi niti poravnani in sosednji, bolj bomo izkoristili pasovno širino. Slika 3.3 prikazuje popolnoma poravnan dostop 32 niti do globalnega pomnilnika, kjer vsaka nit dostopa do 32 zaporednih lokacij. Dostopi niti so v istem segmentu, zato se lahko celoten prenos opravi v eni transakciji [17].



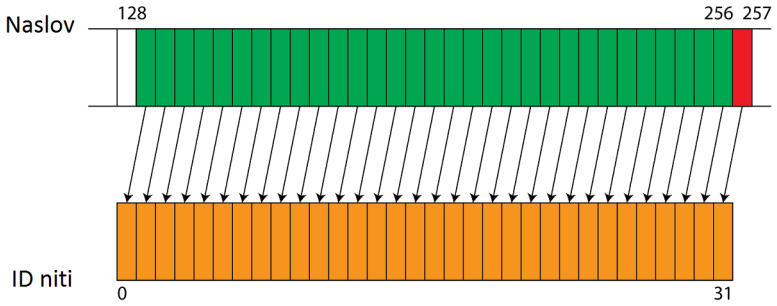
Slika 3.3: Primer popolnoma poravanega dostopa

Novejše GPE niso več tako striktne glede takih poravnav, saj lahko zaradi boljših tehnologij in predpomnilnikov bolje optimizirajo dostope. Četudi dostopi niti niso poravnani, so še vedno v istem segmentu, zato se lahko prenos opravi v eni transakciji.



Slika 3.4: Primer nezaporedni dostopov niti v istem segmentu

Pomnilniška prepostnost se precej zmanjša, če dostopi niso poravnani. Slika 3.5 prikazuje primer, ko so dostopi niti zaporedni, vendar sta zaradi odmika potrebni 2 transakciji, pri čemer je ena zelo slabo izkoriščena, saj je uporabljen zgolj en podatek.



Slika 3.5: Primer neporavnanih dostopov

3.4.2 Lokalni pomnilnik

Lokalni pomnilnik OpenCL v svoji osnovi služi komunikaciji in sinhronizaciji niti v skupini. Niti v delovni skupini komunicirajo in si delijo podatke z uporabo lokalnega pomnilnika, poleg tega pa na nekaterih arhitekturah omogoča skrivanje pomnilniške latence.

Arhitekture kot so GPE, omogočajo optimizacijo pomnilniških dostopov z uporabo lokalnega pomnilnika v pomnilniški hierarhiji OpenCL. OpenCL se namreč na dejanski fizični nivo preslika kot eksplicitni predpomnilnik. Programer lahko tako prenese del podatkov iz globalnega pomnilnika v lokalni pomnilnik in s tem precej zmanjša latenco, saj se lahko dostopi opravijo po posameznih besedah, poleg tega pa so lahko dostopi niti neporavnani, ne da bi poslabšali učinkovitost izvajanja. Glede izbrane velikosti lokalnega pomnilnika na GPE je pomembno predvsem to, da naj ne bo prevelik, saj v slednjem primeru zmanjšamo število skupin, ki se lahko hkrati izvajajo na računski enoti.

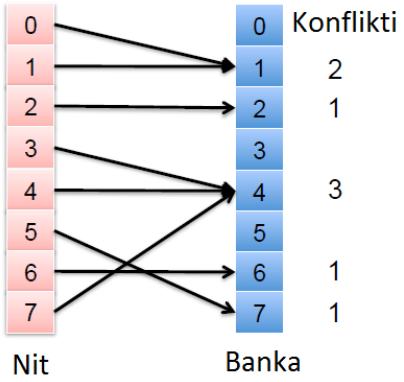
Če želimo kar se da dobro izkoristi lokalni pomnilnik na GPE, moramo

upoštevati še fizično organizacijo [18]. Zaradi sočasnih dostopov, je lokalni pomnilnik razdeljen na banke, tipično na besede velikosti 4 B, pri čemer zaporedne besede pripadajo zaporednim bankam. Za posamezno banko je značilno, da lahko ustreže le enemu dostopu na enkrat. V kolikor so dostopi niti v snopu sosednji in zgolj ena nit dostopa do posamezne banke to ne predstavlja težave. Če pa imamo več niti v snopu, ki dostopajo do iste banke, pa se pojavi konflikt v dostopu do banke (ang. bank conflict), niti morajo v tem primeru počakati, da pridejo na vrsto za dostop. Oba primera sta prikazana na sliki 3.6 in 3.7. Število bank je tipično 32, čeprav je to precej odvisno od specifikacij GPE. Banko, v katero spada beseda, izračunamo iz naslova besede po modulu števila vseh bank. Konflikt stopnje n se zgodi, ko n niti dostopa do iste banke.

Slika 3.6: Lokalni pomnilnik GPE - primer poravnanega in neporavnanega dostopa

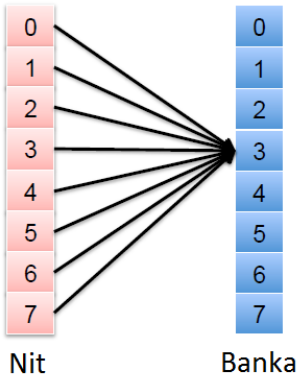
Niti na sliki 3.6 dostopajo do lokalnega pomnilnika GPE brez konfliktov. Na levi sliki so dostopi niti poravnani, na desni pa ne. Na obeh primerih je dostop popolnoma učinkovit, saj naključni dostopi ne vplivajo na dostopni čas lokalnega pomnilnika na GPE.

Na sliki 3.7 imamo primer, ko več niti v snopu dostopa do iste banke v lokalnem pomnilniku. V tem primeru morajo niti, ki dostopajo do iste banke počakati, da pridejo na vrsto.



Slika 3.7: Lokalni pomnilnik GPE - primer konfliktnega dostopa različnih stopenj

Poseben primer konflikta imamo na sliki 3.8, ko vse niti dostopajo do istega naslova. V tem primeru se lahko dostop optimizira z uporabo raztrosa (ang. broadcast) in niti ne čakajo. Podoben primer je tudi sočasen dostop več niti do iste besede v lokalnem pomnilniku. Modernejše enote znajo ta problem rešiti z uporabo razpošiljanja (ang. multicast).



Slika 3.8: Lokalni pomnilnik GPE - uporaba raztrosa

Konflikti v dostopu do banke se rešujejo z uporabo pametnih vzorcev za dostop do lokalnega pomnilnika. Tipično obsegajo uporabo odmikov (ang.

padding), s čimer zamaknemo elemente za določen odmik naprej. Posledično pomeni, da porabimo več lokalnega pomnilnika, pri čemer je majhen del neuporaben. Enostaven primer konflikta bi bil primer, ko vsaka nit dostopa do lokacije, ki je dvakratnik lokalnemu identifikatorju niti. To se zgodi, ko uporabljamo podatkovne tipe v velikosti 8 bajtov. V takem primeru nastanejo konflikti druge stopnje. Težavo rešimo tako, da raje uporabimo podatkovni tip, v velikost 12 bajtov. To najlažje storimo z uporabo strukture, ki poleg elementa vsebuje še dodatno polje v velikosti 4 bajtov in služi zgolj za odmik, da se preprečijo konflikti.

Taka vrsta optimizacije dostopov do lokalnega pomnilnika z upoštevanjem bank je že precej nizkonivojska, zato se jo zaradi kompleksnosti izogibamo, saj ne prinaša občutnih pohitritev. Upoštevanje bank predstavlja dodatno režijo, saj poveča tako porabo lokalnega pomnilnika kot tudi število računskih operacij. Upoštevanje bank zato morda ne bo pohitrilo izvajanja. Še vedno je nekaj konfliktnih dostopov cenejše od dodatnega računanja in precej cenejše od dostopa do globalnega pomnilnika.

3.4.3 Upoštevanje predpomnilnikov

Upoštevanje predpomnilniške hierarhije lahko precej pohitri izvajanje programov, saj so dostopni časi precej manjši. Veliki in večnivojski predpomnilniki so pomembni predvsem v CPE, kjer imamo na voljo malo jeder, zato latenco prikrivamo tako, da minimiziramo število dostopov do glavnega pomnilnika. Za posamezen nivo predpomnilnika je značilno, da se vanj ne prenese zgolj ena beseda, pač pa cel blok. Če torej želimo dobro izkoristiti predpomnilnike v CPE, morajo biti dostopi do pomnilnika čim bolj sosednji. Temu rečemo princip lokalnosti. Seveda pogosto to ni možno zaradi samega algoritma, sploh, če dostopamo do pomnilnika v večjih korakih ali naključno. Današnje GPE imajo tudi svoj predpomnilnik, kot ga poznamo pri CPE, vendar ta ni tako velik in pomemben kot pri CPE.

OpenCL omogoča poizvedbo naprave s parametrom CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE v funkciji clGetDeviceInfo(). Funkcija vrne veli-

kost bloka, ki se prenese pri dostopu do glavnega pomnilnika.

3.5 Uporaba vektorske enote

OpenCL C ima poleg primitivnih tipov vgrajene tudi vektorske tipe, sestavljenih iz imena primitivnega tipa, ki mu sledi število. Število določa število elementov tega tipa v vektorju. Podpirane velikosti v OpenCL so 2, 3, 4, 8 in 16. Vektor 4 celih števil je tako definiran kot *int*4. OpenCL omogoča tudi dostop do posameznih komponent vektorja. Taka abstraktna predstavitev omogoča prenosljivost med različnimi ukaznimi nabori procesorjev in zagotavlja neodvisnost od pravila tankega/debelega konca. Obenem nudi visokonivojsko upravljanje z vektorskimi tipi, saj definira vektorske operacije med istoležnimi elementi in skrbi za ustrezno poravnavo podatkov v pomnilniku (ang. data alignment) glede na velikost tipa. Uporaba vektorskih tipov ni vezana na napravo, zato jih lahko uporabimo tudi, če jih naprava ne podpira. To je mogoče zaradi prevajanja ščepca v času izvajanja. Prevajalnik v tem primeru prevede program v ustrezne ukaze, ki jih podpira naprava [2]. Uporaba vektorskih ukazov je smotrna na procesorjih, ki podpirajo paralelizem tipa SIMD, če jih želimo dobro izkoristiti. V takem primeru lahko precej pospešijo izvajanje programa, če je program pisan na kožo vektorskim enotam. Ukazi za vektorsko procesiranje so večinoma prisotni v CPE in napravah za digitalno procesiranje signalov, kjer je število procesorjev razmeroma majhno in ima posledično na voljo manjše število niti za sočasno izvajanje. Pri eksplicitni uporabi vektorskih podatkovnih tipov moramo upoštevati kakšno podporo SIMD ukazom podpira ciljna arhitektura. Tu moramo upoštevati velikost registrov, to je širino vektorske enote ter podatkovne tipe, ki jih podpira. Uporaba vektorskih tipov večjih, kot jih podpira naprava, se prevede podobno kot odvijanje zanke (ang. loop unrolling). Včasih lahko malo pohitri programe, vendar pa predstavlja večji pritisk na porabo zasebnega pomnilnika, kar lahko zmanjša število skupin, ki se hkrati izvajajo na računski enoti [1]. Kadar delamo z vektorskimi tipi, je pametno prevajalniku one-

mogočiti implicitno vektorizacijo. To storimo tako, da v ščepcu za ključno besedo __kernel dodamo atribut $\_\_attribute\_\_((vec\_type\_hint(tipn)))$. Ta prevajalniku pove, da je ščepec ročno vektoriziran, v tem primeru za n elementov tipa *tip*.

Na skalarnih napravah uporaba vektorskih tipov ne prinaša veliko prednosti, vendar jih lahko kljub temu uporabimo zaradi večje ergonomičnosti pri programiranju. Na tak način precej enostavno povečamo količino dela na nit. Dodatno lahko uporaba vektorskih tipov poveča pomnilniško prepustnost, saj lahko strojna oprema zmanjša število pomnilniških dostopov. Razlog za to so posebni, širši ukazi tipa naloži/shrani (ang. load/store instructions), s katerimi lahko prenesejo več podatkov na enkrat.

OpenCL prevajalniki imajo močno podporo za implicitno vektorizacijo kode. To pomeni, da znajo sami optimizirati kodo tako, da lahko izkoristijo vektorsko enoto. Tipično to počnejo tako, da združujejo sosednje niti v eno. Zato je pametno, da je velikost skupine večkratnik širine vektorske enote. Z uporabo vektorskih tipov lahko prevajalniku onemogočimo učinkovito optimizacijo kode, zato moramo dobro premisliti preden eksplicitno vektoriziramo kodo. Včasih je morda bolje vektorizacijo kar prepustiti prevajalniku, saj lahko sami poslabšamo zadeve. Eksplicitna vektorizacija za točno specifično napravo lahko precej pospeši izvajanje programa na tej napravi, na račun tega, da bo morda učinkovitost na drugi napravi precej manjša [20, 19].

3.6 Podatki izvajalnega okolja OpenCL

OpenCL nam omogoča, da z uporabo funkcije clGetDeviceInfo() poizvemo o podatkih naprave, kot so število računskih enot, največje število niti v bloku, frekvenca naprave, velikost vektorske enote ipd. Na podlagi teh podatkov lahko bolje prilagodimo ščepec pa izvajanje na točno določeni napravi.

CL_DEVICE_GLOBAL_MEM_CACHE_SIZE Velikost predpomnilnika za globalni pomnilnik v bajtih.

CL_DEVICE_GLOBAL_MEM_CACHELINE_SIZE Velikost bloka, ki se prenaša iz globalnega pomnilnika v bajtih.

CL_DEVICE_GLOBAL_MEM_SIZE Velikost globalnega pomnilnika v bajtih.

CL_DEVICE_LOCAL_MEM_SIZE Velikost lokalnega pomnilnika za vsako računsko enoto v bajtih. Pomembno predvsem napravah, ki podpirajo eksplictni predpomnilnik, kot so GPE.

CL_DEVICE_LOCAL_MEM_TYPE Tip oziroma nivo, kamor se preslika lokalni pomnilnik in je odvisno od tipa naprave. Na GPE je to lokalni pomnilnik naprave (SRAM), na CPE globalni pomnilnik.

CL_DEVICE_PREFERRED_VECTOR_WIDTH_* Priporočljiva dolžina vektorskega tipa za vgrajene skalarne tipe. Dolžina vektorja je definirana kot število skalarnih elementov, ki jih lahko shrani vektor. * predstavlja tip CHAR, SHORT, INT, LONG, FLOAT, DOUBLE.

CL_DEVICE_MAX_CLOCK_FREQUENCY Frekvenca naprave v MHz.

CL_DEVICE_MAX_COMPUTE_UNITS Število računskih enot. To je zelo pomemben podatek, saj določa minimalno število skupin, da lahko izkoristimo vse računske enote na napravi. Vedno je bolje imeti več skupin, kot je število računskih enot, saj le tako dovolj dobro izkoristimo vse računske enote. Če imamo na voljo malo računskih enot, je včasih bolje posamezni niti dodeliti več dela, da zmanjšamo režijo preklopa med skupinami.

CL_DEVICE_MAX_WORK_GROUP_SIZE Največje število niti v skupini, ki jo še lahko podpira naprava. Največje število niti, ni vedno optimalno, zato potrebno je najti kompromis med številom niti in skupin.

CL_DEVICE_TYPE Tip naprave kot je CPE, GPE ali pospeševalnik.

Seveda je učinkovitost izvajanja povezana tudi s samim ščepcem. OpenCL omogoča, da poizvemo nekaj podatkov o izvajanju ščepca, kot so največje število niti, ki jih ščepec podpira, priporočen večkratnik velikosti skupine ipd.

CL_KERNEL_WORK_GROUP_SIZE Največje število niti, ki jih še podpira ščepec. Število morda ne bo enako največjemu številu niti, ki jih podpira naprava.

CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE Priporočen večkratnik števila niti v skupini.

CL_KERNEL_LOCAL_MEM_SIZE Velikost uporabljenega lokalnega pomnilnika na skupino. Skupaj z velikostjo lokalnega pomnilnika vpliva na število skupin, ki se lahko sočasno izavajajo na računski enoti.

CL_KERNEL_PRIVATE_MEM_SIZE Velikost zasebnega pomnilnika, ki ga za izvajanje potrebuje vsaka nit. Tudi to ima velik vpliv na število skupin, ki se lahko sočasno izvavajo na računski enoti.

Poglavje 4

Testni problemi

V tem poglavju bomo predstavili probleme, ki predstavljajo praktični del diplomske naloge. Za vsak problem bomo podali opis problema in osnovno idejo za paralelizacijo.

4.1 Histogram

Prvi program, ki ga bomo prilagodili za naše sisteme, je histogram vrednosti med 0 in 255. Imamo dve tabeli, in sicer tabelo vhodnih vrednosti za analizo ter tabelo v velikosti 256 elementov za histogram. Histogram je na začetku v celoti inicializiran na 0. Sekvečni algoritem deluje tako, da iz tabele vhodnih vrednosti zaporedno prebere vse vrednosti in nato glede na to vrednost inkrementira ustrezno polje v hitstogramu.

Program je bolj enostaven in ima malo aritmetičnih operacij, zato pa je zelo odvisen od pomnilniških omejitev kot je prepustnost in izkoriščenost morebitnih predpomnilnikov. Čim bolje je treba izkoristiti pomnilniško hierarhijo tako, da optimiziramo dostope do pomnilnika, pri čemer imamo v mislih minimizacijo števila dostopov do glavnega pomnilnika in upoštevanje principa lokalnosti.

Najbolj enostavna rešitev bi bila, da bi nit i prebrala podatek i iz globalne tabele in nato atomično inkrementirala ustrezno vrednost histograma, ki se

nahaja v globalnem pomnilniku. Branje iz globalne tabele je učinkovito tako na CPE kot GPE, saj nit berejo zaporedno. S tem se na CPE dobro izkoristijo predpomnilniki, na GPE pa visoka pasovna širina in veliko število niti. Največja težava nastane pri redukciji rezultatov v histogram, ker je ta v globalnem pomnilniku. Posledica tega je, da so atomični dostopi precej dragi, saj so precej naključni, kar je še posebej slabo za GPE, kjer morajo biti dostopi do glavnega pomnilnika poravnani. Z uporabo lokalnega pomnilnika bi zmanjšali ceno atomičnih operacij. Na GPE tako ni več pomembno, da so dostopi naključni, na CPE pa bi bolje izkoristili predpomnilnik, saj bi se s tem znebili sinhronizacije med predpomnilniki zaradi dostopov do globalnega pomnilnika. Uporaba lokalnega pomnilnika poveča pomnilniško režijo, saj se morajo delni rezultati na koncu shraniti še v globalni pomnilnik. Vprašanje, ki se tu postavi, je, ali lahko uporaba lokalnega pomnilnika kljub temu naredi izvajanje bolj učinkovito.

4.2 Množenje matrik

Množenje dveh matrik je izvedljivo le, če je število stolpcev prve matrike enako številu vrstic druge matrike. Če je prva matrika dimenzij M*P in druga P*N, potem je rezultat množenja matrika dimenzije M*N. V osnovnem algoritmu vrednost elementa v vrstici i in stolpcu j izračunamo tako, da zmnožimo istoležne elemente v vrstici i prve matrike s stolpcem j druge matrike. Postopek se ponovi za vsak indeks v matriki. Časovna zahtevnost algoritma je $O(n^3)$. Obstajajo sicer tudi boljši algoritmi, vendar je to že izven okvirjev diplomske naloge. Za učinkovito izvajaje je potrebno izkoristiti lastnosti, kot so vektorske enote. Opazimo tudi veliko količino ponovno uporabljenih podatkov, zato je pametno uporabiti še lokalni pomnilnik in čim bolje izkoristiti predpomnilnik.

Zaradi lastnosti problema, se bo ščepec izvajal v dveh dimenzijah. Prva dimenzija bo predstavljala dolžino, druga pa širino končne matrike. S tem bolje izkoristimo morebitne predpomnilnike in zmanjšamo število dostopov

do glavnega pomnilnika. Izvajalno okolje OpenCL namreč najprej izvede ščepec po prvi dimenziji in nato po drugi.

Osnovni algoritem je precej neučinkovit. Najprej na podlagi indeksa i prve in indeksa j druge dimenzije ugotovimo, kateri element računamo, nato pa z zanko zmnožimo istoležne elemente iz vrstice i prve matrike in stolpca j druge matrike ter jih seštejemo, na koncu pa rezultat shranimo v končno matriko. Neučinkovitost se kaže v neporavnanem dostopu do druge matrike, saj dostopamo do elementov v različnih vrsticah, kar pomeni slabo izkoriščenost predpomnilnikov.

4.3 Predponska vsota

Predponska vsota (ang. prefix sum) je algoritem, ki iz danega vhodnega zaporedja števil tvori novo zaporedje števil, kjer je vsako število izračunano kot vsota vseh predhodnikov vhodnega zaporedja števil. [7] Poznamo dve vrsti takega algoritma, in sicer inkluzivno ter ekskluzivno vsoto. Razlika med njima je, da prvi sešteje vse elemente vključno z zadnjim elementom zaporedja, drugi pa ne. V diplomski nalogi bo obravnavana ekskluzivna vsota. To pomeni, da bo v končnem zaporedju prvi element 0, kar je identiteta pri operaciji seštevanja. Algoritem ni omejen na seštevanje, uporabimo lahko tudi kakšno drugo asociativno binarno operacijo:

Definicija: 1 *Ekskluzivna predponska operacija je definirana kot asociativna binarna operacija $\oplus$, ki kot vhodne argumente prejme identiteto I ter tabelo z n elementi*

$$[x_0, x_1, ..., x_{n-1}] \tag{4.1}$$

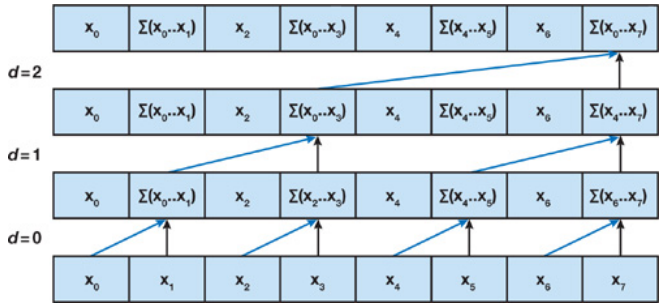
in vrne tabelo

$$[I, x_0, (x_0 \oplus x_1), ((x_0 \oplus x_1 \oplus x_2), ..., (x_0 \oplus x_1 \oplus ... \oplus x_{n-2})] \tag{4.2}$$

Algoritem ima širok spekter uporabe, od sortiranja, primerjave nizov, podatkovnih struktur kot so grafi in drevesa.

Čeprav je predponska vsota na prvi pogled povsem sekvenčne narave, obstaja zelo učinkovit postopek z uporabo vzorca, ki se velikokrat pojavi v paralelnih algoritmih. To so uravnotežena drevesa. Ideja je, da iz vhodnih podatkov zgradimo uravnoteženo binarno drevo in se nato sprehodimo do korena in nazaj ter spotoma računamo predponske vsote. Za uravnoteženo binarno drevo z n listi je značilno, da ima $\log_2 n$ nivojev, med katerimi ima vsak nivo d največ 2^d listov. Drevo, ki ga zgradimo je zgolj koncept, s katerim nitim določimo, kaj morajo storiti na posameznem nivoju drevesa [7].

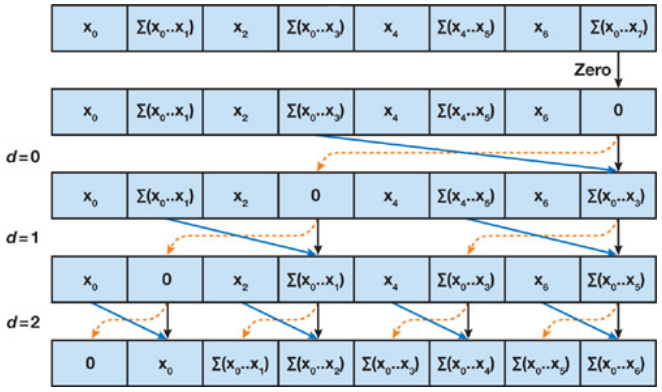
Algoritem je sestavljen iz dveh delov. Prvi del je faza redukcije (ang. reduce phase), kjer se sprehodimo od listov do korena in računamo delne vsote, pri čemer se vsako iteracijo število niti zmanjša za polovico. Taki fazi pravimo paralelna redukcija. V vsaki iteraciji koren poddrevesa postane vsota obeh listov, na koncu faze pa zadnji element tabele, to je koren celotnega drevesa, vsebuje vsoto vseh števil. Celotno fazo na 8 elementih predstavlja slika 4.1. Pri vsaki iteraciji je treba postaviti prepreko, zato, da se niti uskladijo in ne prepisujejo rezultatov.



Slika 4.1: Faza redukcije (vir: GPU Gems 3)

V drugi fazi pometamo navzdol (ang. down-sweep phase). Naprej nit 0 postavi zadnji element, to je koren drevesa na 0, nato pa se sprehodimo od korena navzdol. Začnemo z eno nitjo, nato pa se v vsaki iteraciji število niti podvoji. V vsaki iteraciji desni sin niti v drevesu postane seštevek korena in levega sina, levi sin pa dobi staro vrednost korena. Na koncu faze mora biti prvi element 0, zadnji pa seštevek vseh števil. Celotno drugo fazo na 8

elementih predstavlja slika 4.2. Tudi tu moramo ob vsaki iteraciji postaviti prepreko.



Slika 4.2: Faza pometanja navzdol (vir: GPU Gems 3)

Zgornji algoritem ima še eno težavo. Število elementov, ki jih lahko obdela, je omejeno z dvakratnikom števila niti v skupini. Da bi omogočili poljubno število elementov, razdelimo tabelo v skupine tako, da lahko vsako skupino obdelamo z zgoraj omenjenim algoritmom, skupno vsoto skupine pa shranimo v dodatno tabelo, ki hrani skupno vsoto posameznih skupin. Nato z istim algoritmom obdelamo še tabelo vsot in tako dobimo tabelo vsot za posamezno skupino. Vsaka skupina nato glede na svoj indeks vsem svojim elementom prišteje dobljeno vsoto [8]. Naša implementacija algoritma je torej rekurzivna, vendar to ne predstavlja težave, saj se rekurzija izvaja na gostitelju.

Za pravilno delovanje algoritem potrebuje število niti, ki je potenca števila 2. To bi lahko bila težava za zadnjo skupino, saj velikost tabele morda ne bo deljiva s takim številom. Težavo enostavno odpravimo tako, da dostope niti izven meje tabele enostavno obravnavamo kot vrednost 0, oz. povečamo kapaciteto tabele. To seveda pomeni, da mora zadnja skupina opraviti še nekaj nekoristnega dela. Algoritem uporablja tudi več preprek, zato je pametno ustvariti manjše skupine. Uporaba prepreke povzroči sinhronizacijo lokalnega in zasebnega pomnilnika, saj mora sistem shraniti stanje vseh niti

v skupini, kar predstavlja dodatno časovno režijo [1].

4.4 Problem n teles

Problem n teles je fizikalni problem, pri katerem računamo pot in hitrost gibanja telesa i glede na interakcije sil vseh ostalih teles. V osnovi je problem dovolj splošen, da lahko algoritem uporabimo povsod, kjer med različnimi telesi deluje sila, od interakcij atomov do interakcij nebesnih teles. V tem primeru računamo interakcijo med nebesnimi telesi preko Newtonove gravitacijske sile. Gre predvsem za računsko zahteven problem, saj je gibanje vsakega telesa odvisno od položajev ostalih teles. Za razlago lahko problem poenostavimo na sistem dveh teles i in j v kartezičnem koordinatnem sistemu, med katerima deluje sila F_{ij}, ki je za obe telesi enaka, a nasprotno usmerjena. Naj bosta m_i in m_j masi teles na razdalji dr, κ pa gravitacijska konstanta. Sila med njima je tedaj

$$F_{ij} = \kappa * \frac{m_i * m_j}{dr^2}, \tag{4.3}$$

kjer je razdalja definirana kot

$$dr^2 = dx^2 + dy^2 + dz^2. \tag{4.4}$$

Izračunamo pospešek za telo

$$a_i = \frac{F_i}{m_i} \tag{4.5}$$

ter izračunamo novo hitrost telesa za vse komponente.

$$\begin{aligned} v_{ix}(t + \Delta t) &= v_{ix}(t) + a_{ix}\Delta t \\ v_{iy}(t + \Delta t) &= v_{iy}(t) + a_{iy}\Delta t \\ v_{iz}(t + \Delta t) &= v_{iz}(t) + a_{iz}\Delta t \end{aligned} \tag{4.6}$$

Na koncu pa še nov položaj telesa za vse komponente.

$$\begin{aligned} x_{ix}(t + \Delta t) &= x_{ix}(t) + v(t)\Delta t + \frac{1}{2}a_{ix}\Delta t^2 \\ y_{iy}(t + \Delta t) &= y_{iy}(t) + v(t)\Delta t + \frac{1}{2}a_{iy}\Delta t^2 \\ z_{iz}(t + \Delta t) &= z_{iz}(t) + v(t)\Delta t + \frac{1}{2}a_{iz}\Delta t^2 \end{aligned} \tag{4.7}$$

V primeru več teles moramo samo še upoštevati dejstvo, da je sila na izbrano telo enaka vsoti prispevkov vseh ostalih teles na izbrano telo za posamezno komponento.

$$F_{ix} = \sum_{j,j\neq i} F_{ijx} \qquad F_{iy} = \sum_{j,j\neq i} F_{ijy} \qquad F_{iz} = \sum_{j,j\neq i} F_{ijz} \tag{4.8}$$

Paralelizacija algoritma je enostavna, ustvarimo toliko niti, kolikor je teles, tako, da vsaka nit računa sile za svoje telo. Težava, ki nam jo prinaša OpenCL, je, da niti iz različnih skupin ne moremo sinhronizirati med seboj. Edini način, da sinhroniziramo vse skupine, je zagon novega ščepca, kar seveda predstavlja manjšo režijo. Zaradi tega moramo nekako poskrbeti, da niti svojih rezultatov o koordinatah ne shranijo prehitro, saj je vsako telo v trenutku t odvisno od pozicij ostalih teles v trenutku $t - \Delta t$. Težavo rešimo z uporabo dodatnih tabel, kamor se shranjujejo novi rezultati, izvajalno okolje OpenCL pa poskrbi za zamenjavo kazalcev pred ponovnim zagonom ščepca.

4.5 Bitonično urejanje

Bitonično urejanje je algoritem za urejanje števil, zasnovan prav posebej za paralelna okolja, kjer moramo za optimalno izvajanje izkoristiti večjedrnost sistema. Urejanje z bitonično sekvenco deluje tako, da naprej poljubno sekvenco pretvori v bitonično sekvenco, nato pa jo uredi po velikosti. Algoritem deluje samo na tabelah velikosti 2^k (obstajajo variante brez te omejitve, vendar niso predmet diplomske naloge). Časovna zahtevnost algoritma je ne glede na vhodno urejenost podatkov vedno enaka.

Dani sta dve zaporedji števil A in B. Zaporedje A naj bo monotono naraščajoče, B monotono padajoče. Naj novo bo zaporedje C konkatenacija teh dveh zaporedij. Zaporedje C je sestavljeno iz dveh monotonih zaporedij, pravimo, da je bitonično. Za bitonično sekvenco je značilno, da se lahko naklon v celotnem zaporedju spremeni samo enkrat.

Definicija: 2 *Bitonično zaporedje je zaporedje, za katerega velja, da*

$$x_0 \leq x_1, ... \leq x_k \geq ... \geq x_{n-1} \tag{4.9}$$

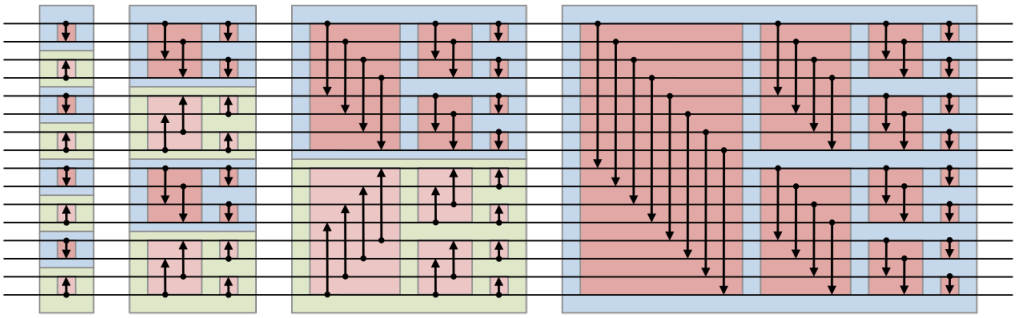
za nek indeks $k, 0 \leq k < n$. Vsako zaporedje, ki ga lahko s ciklično rotacijo spremenimo v bitonično, je prav tako bitonično [5].

Fazo urejanja bitoničnega zaporedja imenujemo zlivanje. Operacija najprej primerja vsak element iz prve polovice (indeksi od 0 do $n/2-1$) s soležnim elementom iz druge polovice (indeksi od $n/2$ do $n-1$). Če je element iz spodnje polovice večji, potem ju zamenja. V primeru zaporedja C, dobimo novo zaporedje D, ki sicer ni bitonično, sta pa zato obe polovici. Vsak element v zaporedju D iz prve polovice, je manjši ali enak vsem elementom iz druge polovice. Postopek se rekurzivno nadaljuje nad obema polovicama zaporedja D. Pri vsakem razbitju manjši elementi potujejo levo, večji pa desno. V zadnjem koraku razbijemo tabele v velikosti 2 elementov. Zaporedje je sedaj urejeno [6].

Seveda mora biti zaporedje bitonično, da ga lahko uredimo. Pretvorba poljubnega zaporedja v bitonično sekvenco je sestavljena iz več faz, pri čemer po vsaki fazi dobimo novo lokalno zaporedje, ki je urejeno. V prvi fazi urejamo pare, kjer vsak lih par uredimo naraščajoče, vsak sodi par pa padajoče. Zatem urejamo lokalna zaporedja velikosti 4, kjer najprej prvi element primerjamo s tretjim in drugega s četrtim, nato pa še prvega z drugim in tretjega s četrtim in nato ponovimo prehoden postopek, kjer smo urejali pare. Lihe četverke urejamo naraščajoče, sode padajoče. Iz tega sklepamo vzorec za poljubne višje stopnje, kjer število urejenih elementov narašča eksponentno glede na stopnjo. Najprej se vsak element iz spodnje polovice primerja s soležnim elementom iz zgornje polovice, nato pa ponovimo vse postopke iz nižjih stopenj, smer urejanja pa določa sodost in lihost n-terk, ki jih urejamo. Celoten postopek na 16 elementih, je vizualno predstavljen na sliki 4.3, kjer prve tri faze urejajo poljubno zaporedje v bitonično, nato pa sledi operacija zlivanja, kjer se elementi dokončno uredijo.

Če bi želeli zamenjati vrstni red urejanja, moramo zgolj obrniti smer primerjanja. Puščice na zgornji sliki bi bile obrnjene ravno obratno.

Prednost bitoničnega urejanja je, da je število operacij vedno isto, ne glede na urejenost vhodne tabele. Ta neodvisnost operacij naredi bitonično



Slika 4.3: Bitonično urejanje (vir: Wikipedia)

urejanje izrazito paralelno usmerjeno. Osnovna ideja za paralelizacijo, da ustvarimo toliko, niti kot je elementov. Vsaka nit glede na fazo in iteracijo primerja z ustreznim elementom, pri čemer v vsaki iteraciji polovica niti ne naredi ničesar, sicer bi isto primerjavo opravili dvakrat. Zelo pomembno je, da se vsaka iteracija konča pred začetkom druge, sicer urejanje ne bo pravilno. Zatorej moramo po vsaki iteraciji sinhronizirati vse niti. OpenCL ne omogoča globalne sinhronizacije niti izven skupin, zato je edina možnost, da po vsaki iteraciji ponovno zaženemo ščepec z ustrezno nastavljenimi parametri, ki določajo fazo in iteracijo.

Poglavje 5

Rezultati

5.1 Opis izbranih arhitektur

V nadaljevanju diplomske naloge si bomo pogledali nekaj testnih programov, ki jih bomo prilagodili, da se bodo čim bolj učinkovito izvajali na izbranih arhitekturah. Kot smo že omenili, imajo različne arhitekture različne prednosti in slabosti na določenih področjih. Ugotoviti je potrebno, katere podatke o platformi uporabiti, da se bo program na izbrani platformi izvajal čim bolj učinkovito. Tu gre predvsem za število računskih enot, število delovnih skupin in število niti v posamezni skupini. Upoštevati moramo tudi različno pomnilniško hierarhijo, kot je uporaba lokalnega pomnilnika, kjer je na voljo, oziroma napisati program tako, da bo dobro izkoristil predpomnilnik, kjer lokalnega pomnilnika ni. Druga stvar je uporaba SIMD ukazov. Kako pametno napisati program, da te ukaze uporablja, če so na voljo in kako postopati, če jih ni.

V nadaljevanju sledi opis posameznih sistemov za katere bomo optimizirali programe in analizirali izvajanje programov z različnimi nastavitvami in na podlagi tega ugotovili kateri podatki o platformi koristijo pri optimizaciji ter kako jih uporabiti. Za podlago pri optimizaciji programov bomo uporabili informacije iz prejšnjih poglavij.

5.1.1 Sistem 1: Intel Core i5-2450M

Prvi sistem predstavlja običajen prenosnik, z dvojedrnim procesorjem Intel Core i5 arhitekture x64, pri čemer ima vsako jedro strojno podporo za 2 niti, frekvenca posameznega jedra pa je 2,5 GHz. Za izvajanje ima sistem na voljo 6 GB pomnilnika in 3 MB predpomnilnika na nivoju L3, 256 kB na nivoju L2 ter na nivoju L1 še 32 kB za ukaze in 32 kB za podatke. Iz podatkov pri OpenCL poizvedbi izvemo, da ima CPE 4 računske enote, kar ustreza številu niti, podporo za do 4 vektorske operacije v plavajoči vejici ter podporo za največ 1024 niti v delovni skupini. Intel za čim bolj učinkovito implicitno vektorizacijo priporoča, da naj bo velikost skupine večkratnik števila 8. Omeniti velja še podporo za AVX (ang. Advanced Vector Extensions) ukaze. To so 256 bitni vektorski ukazi, namenjeni izključno operacijam v plavajoči vejici. Celoten sistem poganja operacijski sistem Windows 10.

5.1.2 Sistem 2: Intel Xeon Phi 5110P

Intel Xeon Phi je Intelov odgovor na današnje zmogljive grafične kartice, saj predstavljajo resno konkurenco predvsem v znanstvenem svetu, kjer se računanje pri mnogih matematičnih, fizikalnih, kemijskih problemih zelo pohitri, z uporabo velikega števila procesorjev, ki lahko hkrati neodvisno izvajajo svoj del naloge. Xeon Phi 5110P je simetrični multiprocesor na enem čipu arhitekture MIC (ang. Many Integrated Core). Skupno ima 60 jeder, zato se uvršča med mnogojedrnike, pri čemer vsako jedro deluje s frekvenco 1 GHz in ima strojno podporo za 4 niti. Frekvenca zaradi velikega števila jeder ni tako pomembna kot pri običajih CPE, zato je lahko precej nižja. Poraba električne energije je tako precej manjša in s tem tudi segrevanje. Veliko število jeder pomeni, da lahko zelo hitro naletimo na omejitve s prepustnostjo pomnilnika. Xeon ta problem rešuje z visoko zmogljivo dvosmerno povezavo med jedri ODI (ang. On Die Interconnect) ter 8 pomnilniškimi krmilniki za dvosmerno hitro komunikacijo s pomnilnikom GDDR5. Sistem ima za izvajanje na voljo 6 GB pomnilnika. Za prikrivanje latence skrbijo tudi veliki

predpomnilniki. Vsako jedro je opremljeno s 512 kB predpomnilnika na nivoju L2 (skupaj torej 30 MB L2) in na nivoju L1 še 32 kB za ukaze in 32 kB za podatke. Iz podatkov pri OpenCL poizvedbi izvemo, da imamo na voljo $59 \cdot 4 = 236$ računskih enot (1 procesor ima za komunikacijo med gostiteljem in koprocesorjem) in podporo za največ 8192 niti v delovni skupini. Xeon ima tudi močno podporo za izvajanje vektorskih operacij, saj ima 512 bitno vektorsko enoto, ki omogoča hkratno obdelavo do 8 elementov v dvojni in 16 v enojni natančnosti. Za bolj uspešno implicitno vektorizacijo je bolje, če je število niti večkratnik števila 16. Prevajalnik bo v tem primeru združeval niti tako, da bo kar najbolje izkoristil vektorsko enoto. Iz števila računskih lahko sklepamo, da je potrebno za osnovno izkoriščenost potrebnih vsaj 236 skupin, za optimalno pa se priporoča vsaj 1000 skupin. Skupina mora biti dovolj velika, da obenem zmanjšamo učinek režije preklapljanja med skupinami [23]. Na gostitelju teče operacijski sistem CentOS 6.4.

5.1.3 Sistem 3: Nvidia Tesla K20

Nvidia Tesla K20 je grafična kartica namenjena predvsem v znanstvenem svetu, saj zaradi svojih specifikacij zelo draga v primerjavi z običajnimi karticami. Ima 2496 procesnih jeder, s frekvenco 705 MHz, ki so razdeljena po 13 računskih enotah. Vsaka računska enota ima tako 192 jeder. Za izvajanje ima sistem na voljo 5 GB GDDR5 pomnilnika skupaj s 1,5 MB predpomnilnika na nivoju L2, vsaka računska enota pa ima še 49 KB lokalnega pomnilnika za sinhronizacijo znotraj skupine ter prikritje visoke latence. Za visoko prepustnost z globalnim pomnilnikom skrbi 320 bitno vodilo. Tesla K20 je skalarne narave, zato vektorizacija ni potrebna. Za čim boljšo skalabilnost in prikrivanje latence se priporoča čim več skupin, prav tako tudi čim večjo velikost skupine, ki naj bo večkratnik števila 32, da čim bolje izkoristimo izvajanje po snopih . K20 podpira največ 1024 niti na delovno skupino [24]. Na gostitelju teče operacijski sistem CentOS 6.4.

5.2 Opis merjenja časa izvajanja

Za vse ščepce smo merili čas izvajanja programa pri različnih velikostih skupine. Tako smo želeli ugotoviti, kako velikost skupine vpliva na čas izvajanja, kolikšen in kakšen vpliv ima velikost skupine na izvajanje, kdaj je bolje uporabiti več niti, kakšna je razlika med arhitekturami glede števila niti v skupini. Pri optimizaciji programov smo uporabili različne podatke, kot so število računskih enot, maksimalno število niti na skupino, priporočen večkratnik števila niti na skupino, podatke o morebitnih vektorskih enotah ter predpomnilniško hierarhijo in lokalni pomnilnik.

Za merjene časa izvajanja smo uporabili funkcijo $ftime()$, ki omogoča merjenje časa na milisekundo natančno. Večino meritev smo ponovili od trikrat do petkrat, odvisno od časa izvajanja in na podlagi vseh meritev izračunali tudi standardni odklon, ki služi za oceno napake pri meritvah. Če je bil čas izvajanja predolg, smo meritev opravili samo enkrat ali dvakrat, saj napake niso več tako pomembne. To se je lahko zgodilo v primeru, ko smo uporabili zgolj eno nit na skupino, ali pa ko je bil problem enostavno prevelik.

Iz raziskovalnih razlogov smo uporabili tudi zelo majhne skupine, pri čemer pa smo na GPE Tesla K20 pazili, da je število niti večkratnik števila 32, da smo vedno polno izkoristili izvajanje po snopih.

5.3 Histogram

Za vse tri sisteme smo merili čas izvajanja pri različnem številu niti in več možnih rešitvah za velikost problema $N = 3 \cdot 10^8$.

5.3.1 Rešitev za sistem 1

Pri enostavni rešitvi z globalnim pomnilnikom, smo pri 128 nitih na skupino izmerili čas 2 sekund. Število niti v skupini v tem ni tako bistvenega pomena, saj večje skupine ne prinašajo pohitritev, kakor je razvidno iz tabele 5.1. V tabeli opazimo, da imamo pri enostavnem algoritmu težave, če uporabimo 256 niti ali več, saj testi pokažejo, da algoritem ne deluje pravilno. Razlog je najverjetneje napaka v samem ogrodju OpenCL.

Za CPE velja, da ima malo število računskih enot, torej bi veljajo premisliti o zmanjšanju števila skupin. Poleg tega bi za delne rezultate histograma uporabili še lokalni pomnilnik, saj bi s tem zmanjšali ceno atomičnih operacij in komunikacijo med predpomnilniki. Taka rešitev ima težavo, ker precej poveča pomnilniško režijo, saj se morajo rezultati na koncu shraniti še v globalni pomnilnik. To kažejo tudi meritve prikazane v tabeli 5.1, saj z uporabo večjih skupin precej zmanjšamo čas izvajanja.

To napelje do končne rešitve. Število skupin in posledično pomnilniško režijo pri shranjevanju rezultatov v globalni pomnilnik zmanjšamo tako, da posamezni niti dodelimo več dela. Vsaka nit tako prebere k zaporednih elementov, pri čemer vsaka nit začne z določenim odmikom. Odmik izračunamo tako, da globalni indeks niti pomnožimo s k. Zaporedno branje podatkov poskrbi tudi za dobro izkoriščenost predpomnilnikov. Rezultati tabele 5.2 kažejo še dodatno izboljšanje, saj smo skupaj čas izvajanja zmanjšali iz 2 sekund na 0,7 sekunde.

	Čas [s]	
Število niti	Globalni pomnilnik	Lokalni pomnilnik
1	$2,91 \pm 0,17$	-
2	$2,17 \pm 0,07$	-
4	$2,02 \pm 0,04$	-
8	$1,97 \pm 0,00$	-
16	$2,0 \pm 0,02$	-
32	$2,0 \pm 0,01$	$9,31 \pm 0,04$
64	$2,0 \pm 0,01$	$4,83 \pm 0,02$
128	$2,0 \pm 0,01$	$2,68 \pm 0,03$
256	TF	$1,82 \pm 0,02$
512	TF	$1,27 \pm 0,01$
1024	E	$1,01 \pm 0,01$

Tabela 5.1: Rezultati meritev pri skupinah različne velikosti za sistem 1

Število niti	k elementov na nit	Čas [s]
1024	4	$0,77 \pm 0,00$
	8	$0,73 \pm 0,01$
	16	$0,70 \pm 0,01$
	32	$0,70 \pm 0,00$

Tabela 5.2: Rezultati meritev za sistem 1, ko vsaka nit opravi več dela

5.3.2 Rešitev za sistem 2

Rezultati meritev v tabeli 5.3 kažejo, da pri enostavnem algoritmu, število niti v bloku, razen za majhne skupine, nima velikega vpliva na čas izvajanja.

Za bolj učinkovito izvajanje uporabimo enako idejo kot pri sistemu 1, uporabimo lokalni pomnilnik tako, da delne rezultate histograma shranimo v lokalni pomnilnik. Rezultati meritev v tabeli 5.3 kažejo, da dovolj velika skupina lahko preseže ceno redukcije skupin v globalni pomnilnik. Nadaljnje povečanje dela posamezni niti ne bi prineslo bistvenih izboljšav, saj imamo na voljo veliko število računskih enot, zato potrebujemo tudi dovolj skupin.

	Čas [s]	
Število niti	Globalni pomnilnik	Lokalni pomnilnik
1	$3,73 \pm 0,07$	-
2	$3,31 \pm 0,05$	-
4	$3,37 \pm 0,08$	-
8	$3,39 \pm 0,13$	-
16	$3,31 \pm 0,07$	-
32	$3,28 \pm 0,08$	$3,14 \pm 0,06$
64	$3,26 \pm 0,03$	$2,65 \pm 0,12$
128	$3,30 \pm 0,08$	$2,20 \pm 0,07$
256	TF	$2,16 \pm 0,12$
512	TF	$2,13 \pm 0,14$
1024	TF	$1,97 \pm 0,07$
2048	TF	$1,91 \pm 0,07$
4096	TF	$2,02 \pm 0,10$

Tabela 5.3: Rezultati meritev pri skupinah različne velikosti za sistem 2

5.3.3 Rešitev za sistem 3

Z osnovno rešitvijo smo pri 128 nitih na skupino izmerili čas 0,79 sekund. Rezultati meritev v tabeli 5.4 kažejo, da kljub uporabi lokalnega pomnilnika na GPE in uporabi večjih skupin ne pridobimo na hitrosti. Razlog je najverjetneje v tem, da gre že v osnovi za zelo enostaven algoritem, razmeroma poravnanimi dostopi do globalnega pomnilnika, zato dobro izkoristi pomnilniško prepustnost.

	Čas [s]	
Število niti	Globalni pomnilnik	Lokalni pomnilnik
32	$0,78 \pm 0,04$	$0,81 \pm 0,05$
64	$0,78 \pm 0,04$	$0,78 \pm 0,05$
128	$0,79 \pm 0,03$	$0,77 \pm 0,05$
256	TF	$0,78 \pm 0,04$
512	TF	$0,78 \pm 0,05$
1024	TF	$0,78 \pm 0,04$

Tabela 5.4: Rezultati meritev pri skupinah različne velikosti za sistem 3

5.3.4 Sklep

Histogram je zelo enostaven program z veliko V/I in malo računskih operacij. Zaradi svoje enostavnosti se ga sploh ne splača izvajati na GPE in koprocesorju, saj je režija pri prenašanju podatkov na napravo enostavno prevelika v primerjavi z delom, ki ga opravi naprava. Histogram se je najbolje izkazal na običajni CPE. Primer je pokazal, da lahko tudi na CPE z uporabo lokalnega pomnilnika nekoliko izboljšamo učinkovitost. Vektorizacija je prepuščena implicitni vektorizaciji, da lahko ostane velikost problema poljubna. Dodatno smo imeli na vseh arhitekturah težave pri osnovnem algoritmu, če smo uporabili 256 ali več niti, saj imamo težavo s sinhronizacijo atomičnih dostopov do globalnega pomnilnika. Razlog je najbrž napaka v implementaciji ogrodja OpenCL.

5.4 Množenje matrik

Za vse sisteme smo merili čas izvajanja pri različnem številu niti za različne rešitve in ugotavljali, katera rešitev je boljša za specifičen sistem. Za testne namene smo privzeli matriki A in B. Velikost matrike A je 4160x3200, velikost matrike B pa 3200x2400. Za zagotovitev čim bolj natančnih rezultatov in da ne bi prišli do napačnih sklepov, smo na sistemu 2 in sistemu 3 merili čas izvajanja tudi za večji problem, in sicer z matriko C velikosti 6400x16000 ter matriko D velikosti 16000x14400. V nadaljevanju bomo zaradi enostavnosti njujin produkt označili kot AB in CD.

5.4.1 Rešitev za sistem 1

V tabeli 5.5 so rezultati meritev za osnovni algoritem, to je brez uporabe vektorskih enot in drugih trikov. Ta nam služi zgolj za primerjavo, koliko so vredne posamezne izboljšave. Meritve kažejo, da se imamo pri manjših skupinah veliko režijo zaradi preklopov med skupinami, poleg tega pa prevajalnik ne more dobro vektorizirati ščepeca. Pri skupinah večjih od 4x4 se

rezultati meritev ne izboljšujejo več bistveno, kar pomeni, da so procesne enote popolnoma zasedene.

Velikost skupine	Čas [s]
1 x 1	$156,60 \pm 0,30$
2 x 2	$155,77 \pm 0,10$
4 x 4	$39,31 \pm 0,17$
8 x 8	$39,06 \pm 0,02$
16 x 16	$39,05 \pm 0,02$
32 x 32	$39,08 \pm 0,02$

Tabela 5.5: Rezultati meritev za sistem 1 - osnovni algoritem - produkt AB

Če želimo, da bi se algoritem čim bolj učinkovito izvajal, moramo poskrbeti za učinkovite dostope do pomnilnika. Na CPE to rešujemo z uporabo pametnih, čim bolj zaporednih dostopov tako, da maksimiziramo število zadetkov v predpomnilniški hierarhiji. Na žalost to ne velja za matriko B, saj do nje dostopamo po stolpcih. Prva ideja, kako to odpraviti je, da pred množenjem opravimo transformacijo matrike. Transformacija sicer predstavlja določeno časovno režijo, vendar korist, ki jo dobimo od upoštevanja predpomnilniku prijaznih dostopov, v primeru dovolj velike matrike preseže časovno režijo zaradi transformacije. Še ena večja prednost je, da sedaj lahko zelo dobro izkoristimo vektorsko enoto sistema. Informacije o sistemu namreč razkrijejo, da vektorska enota omogoča hkratno obdelavo do 4 operacij v plavajoči vejici enojne natančnosti. Matriko sedaj interpretiramo, kot tabelo tipa $float4$. To pomeni, da vsak stolpec sedaj predstavlja 4 vrednosti. Zaradi hkratnega računanja 4 elementov na nit, je treba za 4-krat zmanjšati globalno velikost dimenzije 0, samodejno vektorizacijo pa izključimo.

Ideja je podobna kot pri običajnem množenju, s to razliko, da sedaj posamezna nit računa 4 zaporedne elemente naenkrat, do druge matrike pa dostopamo po vrstici, saj je transformirana. Posamezna nit (i, j) najprej inicializira vektor 4 vrednosti (spremenljivka tipa $float4$), imenujmo ga akumulator na 0. Ta predstavlja 4 zaporedne indekse matrike, za katere nit računa skalarni produkt. Iz prve matrike nit prenese vektor iz vrstice i ter ga zmnoži

s 4 zaporednimi vektorji v stolpcu j druge matrike, nato pa seštevke množenj shrani v ustrezen indeks v akumulatorju. Postopek ponavljamo, dokler ne zmnožimo cele vrstice ter na koncu shranimo rezultate v akumulatorju v globalni pomnilnik.

Rezultati v tabeli 5.6 kažejo, da smo v primerjavi z osnovnim algoritmom (tabela 5.5), dobili približno 5-kratno pohitritev. Očitno je transformacija druge matrike zelo pripomore k zmanjšanju zgrešitvenih kazni, z vektorsko enoto pa z enim ukazom zmnožimo 4 elemente. Za dobro učinkovitost zadostuje velikost skupine 8x8, večje pa ne prinašajo občutnejših pohitritev. Iz tabele 5.7 lahko razberemo, da uporaba večjih vektorskih tipov, kjer bi posamezna nit obdelala 8 elementov, se navkljub enoti AVX ni izkazala za učinkovitejšo, saj so bili rezultati meritev malo slabši. Razlog za to je, da ima cena transformacije matrike v tem primeru večji vpliv, saj predpomnilnik bolje izkoristimo že z večjo vektorsko enoto.

Težava prejšnjega algoritma je, da moramo pred izvedbo množenja opraviti transformacijo matrike, kar predstavlja določeno časovno režijo. Ugotoviti je treba, ali lahko morda vektorsko množenje opravimo tudi brez transformacije, pri čemer bi posamezna nit še vedno računala 4 elemente na enkrat. Tudi to je možno. Iz prejšnjega primera se vidi, da se posamezna komponenta vektorja iz prve matrike vedno pomnoži z istim indeksom vrstice iz druge matrike. Podobno kot v prejšnjem primeru tudi tokrat delne rezultate shranjujemo v akumulator, vektor 4 vrednosti. Nit (i, j) torej najprej prenese vektor iz vrstice i prve matrike, nato pa posamezno komponento iz tega vektorja pomnoži z istoležnim vektorjem v stolpcu j druge matrike, rezultat množenja pa prišteje akumulatorju. V takem primeru skalarni tip množimo z vektorskim tipom, OpenCL pa v tem primeru sam prevede kodo tako, da bo rezultat pravilen. Postopek ponavljamo, dokler ne zmnožimo cele vrstice ter na koncu shranimo rezultate v akumulatorju v globalni pomnilnik.

Žal se je taka rešitev, kot kaže tabela 5.6, izkazala za precej neučinkovito, saj so bili rezultati meritev, ne glede na velikost bloka, vedno okoli 39 sekund, kar je enako kot pri osnovnem algoritmu. Razlog je verjetno, da je

zgrešitvenih kazni precej več. To se kaže v tem, da so rezultati z uporabo večjega vektorskega tipa precej boljši. Rezultati merjenj v tabeli 5.7 kažejo okoli 2-kratno pohitritev, če posamezna nit obdela 8 elementov. Čas izvajanja je okoli 19,5 sekund, kar pa je še vedno slabše kot v primeru, kjer smo drugo matriko transformirali. Razlog za tako pohitritev je v enoti AVX, ki omogoča obdelavo do 8 števil v plavajoči vejici enojne natančnosti. Taka rešitev je bolj primerna za manjše matrike, kjer se režija pri pretvorbi matrike bolj pozna.

	Čas [s]	
Velikost skupine	Množenje s transformacijo	Množenje brez transformacije
1 x 1	$8,47 \pm 0,10$	$39,78 \pm 0,04$
2 x 2	$8,09 \pm 0,03$	$39,58 \pm 0,02$
4 x 4	$7,95 \pm 0,02$	$39,51 \pm 0,02$
8 x 8	$7,93 \pm 0,04$	$39,48 \pm 0,02$
4 x 16	$7,89 \pm 0,04$	$39,46 \pm 0,00$
8 x 32	$7,92 \pm 0,05$	$39,68 \pm 0,02$

Tabela 5.6: Rezultati meritev za sistem 1 - vektorski tip $float4$ - produkt AB

	Čas [s]	
Velikost skupine	Množenje s transformacijo	Množenje brez transformacije
1 x 1	$9,84 \pm 0,13$	$19,87 \pm 0,06$
2 x 2	$9,22 \pm 0,03$	$19,52 \pm 0,03$
4 x 4	$9,03 \pm 0,02$	$19,54 \pm 0,05$
1 x 8	$8,86 \pm 0,02$	$19,46 \pm 0,01$
2 x 8	$8,92 \pm 0,02$	$19,44 \pm 0,01$
1 x 16	$8,82 \pm 0,01$	$19,46 \pm 0,00$
4 x 16	$8,87 \pm 0,04$	$19,46 \pm 0,02$
4 x 32	$8,93 \pm 0,05$	$19,50 \pm 0,01$

Tabela 5.7: Rezultati meritev za sistem 1 - vektorski tip $float8$ - produkt AB

5.4.2 Rešitev za sistem 2

Tabela 5.8 kaže rezultate meritev za osnovni algoritem. Rezultati merjenja časa izvajanja osnovnega algoritma, pri skupinah različnih velikosti kažejo, da velikost skupine nima velikega pomena razen, kadar je skupina res majhna, saj OpenCL takrat ne more vektorizirati ščepca. Majhne razlike pri večjih velikostih so posledica režije pri preklopu skupin. Kot osnovo za nadaljno primerjavo privzemimo, da je čas množenja matrike AB 2,84 sekund, CD pa 89,66 sekund.

	Čas [s]	
Velikost skupine	AB	CD
1 x 1	$33,21 \pm 0,19$	$2423,70$
2 x 2	$3,08 \pm 0,21$	$89,70 \pm 0,33$
4 x 4	$2,91 \pm 0,11$	$89,59 \pm 0,19$
8 x 8	$2,84 \pm 0,10$	$89,66 \pm 0,11$
16 x 16	$2,81 \pm 0,05$	$90,56 \pm 0,05$
32 x 32	$3,14 \pm 0,15$	$89,70 \pm 0,10$

Tabela 5.8: Rezultati meritev za sistem 2 - osnovni algoritem - produkt AB in CD

Na Xeon Phi moramo dobro izkoristiti veliko število računskih enot in 512 bitno vektorsko enoto, kar konkretno v našem primeru pomeni, da lahko hkrati zmnožimo do 16 elementov v plavajoči vejici enojne natančnosti. Uporabili bomo iste algoritme, kot smo jih pri sistemu 1, s to razliko, da jih bomo prilagodili za izvajanje nad 16 elementi naenkrat. Razlika je zgolj v tem, da se vektorski tip $float4$ spremeni v $float16$ ter temu primerno nastaviti globalno velikost.

Iz tabele 5.9 lahko razberemo, da ideja s transformacijo druge matrike ne izboljša oziroma celo malo poslabša čas izvajanja za produkt AB. To pa ne velja za produkt CD, saj rezultati v tabeli 5.10 kažejo na izboljšanje, saj smo čas izvajanja uspeli zmanjšati na okoli 71 sekund. V obeh primerih pa je razvidno, da z večanjem velikosti skupine, učinkovitosti izvajanja ne izboljšujemo več veliko. Razlog za to je, da vektorska enota nadomešča 16

niti. Posamezna jedra so zato dobro izkoriščena, svoj pomen pa izgubi tudi skrivanje latence z velikim številom niti. Kljub temu ni pametno izbrati premajhne skupine, saj tako zmanjšamo morebiten učinek režije preklapljanja med skupinami.

Ostane še algoritem brez transformacije druge matrike. Rezultati meritev v tabelah 5.9 in 5.10 kažejo, da smo uspeli doseči izboljšanje v primerjavi s prejšnjim algoritmom. To se še posebej pozna za produkt CD, kjer smo čas izvajanja uspeli zmanjšati na okoli 45,5 sekund. Kot kaže, večji vektorski tip, $float16$, zelo dobro izkoristi pomnilniško prepustnost in predpomnilnike ter zmanjša število aritmetičnih operacij, saj z enim ukazom zmnoži 16 elementov. Iz tega lahko sklepamo, da zaradi velike vektorske enote transformacija matrike ne pripomore več tako dobro k učinkovitemu izvajanju. Podobno kot prej večje število niti v skupini ne prinaša bistvenih izboljšav, saj niti nadomešča vektorska enota.

	Čas [s]	
Velikost skupine	Množenje s transformacijo	Množenje brez transformacije
1 x 1	$3,13 \pm 0,08$	$2,34 \pm 0,23$
2 x 2	$3,18 \pm 0,17$	$2,35 \pm 0,25$
1 x 4	$3,00 \pm 0,09$	$2,21 \pm 0,12$
2 x 16	$3,17 \pm 0,31$	$2,37 \pm 0,18$
2 x 32	$3,03 \pm 0,05$	$2,16 \pm 0,16$

Tabela 5.9: Rezultati meritev za sistem 2 - vektorski tip $float16$ - produkt AB

	Čas [s]	
Velikost skupine	Množenje s transformacijo	Množenje brez transformacije
1 x 1	$72,34 \pm 0,21$	$46,77 \pm 0,08$
1 x 16	$71,19 \pm 0,10$	$46,77 \pm 0,12$
2 x 2	$71,79 \pm 0,48$	$46,69 \pm 0,35$
4 x 16	$71,44 \pm 0,31$	$45,47 \pm 0,13$
2 x 32	$71,24 \pm 0,19$	$46,67 \pm 0,30$
4 x 32	$71,45 \pm 0,40$	$45,31 \pm 0,14$

Tabela 5.10: Rezultati meritev za sistem 2 - vektorski tip $float16$ - produkt CD

5.4.3 Rešitev za sistem 3

Rezultati merjenja časa izvajanja osnovnega algoritma v tabeli 5.11, pri skupinah različnih velikosti kažejo, da ima velikost skupine precejšen vpliv na hitrost izvajanja. Algoritem se najbolje izvaja, če uporabimo največjo možno velikost skupine. To je v skladu s pričakovanji, saj je Tesla K20 narejena za sočasno izvajanje velikega števila niti. Te se izvajajo po snopih velikosti 32, zato je izvajanje najbolj učinkovito, če je tudi velikost skupine večkratnik števila 32. Z velikim številom niti Tesla K20 nadomešča vektorsko enoto in prikriva latenco. Kot osnovo za nadaljno primerjavo privzemimo, da je čas množenja produkta AB 0,95 sekund, CD pa 27,25 sekund.

	Čas [s]	
Velikost skupine	AB	CD
4 x 4	$3,14 \pm 0,05$	$130,53 \pm 0,39$
8 x 8	$1,55 \pm 0,06$	$57,74 \pm 0,04$
16 x 16	$1,09 \pm 0,05$	$35,63 \pm 0,09$
32 x 32	$0,95 \pm 0,05$	$27,25 \pm 0,05$

Tabela 5.11: Rezultati meritev za sistem 3 - osnovni algoritem - produkt AB in CD

Za bolj učinkovito izvajanje moramo izkoristiti lokalni pomnilnik naprave. Težava, s katero se srečujemo pri osnovnem algoritmu, je visoka latenca globalnega pomnilnika, pri čemer imamo tudi veliko ponovne uporabe istih

podatkov. Uporaba lokalnega pomnilnika je zato nuja. Ker pa je lokalni pomnilnik precej majhen, moramo problem razdeliti na več manjših delov.

Osnovna ideja je, da matriko razbijemo na manjše kvadratne ploščice enake velikosti. Te ploščice določajo, kako se v lokalni pomnilnik prenašajo podatki. Fizično torej lokalni pomnilnik predstavljata dve matriki v velikosti WxW, pri čemer W predstavlja velikost skupine za obe dimenziji. V tem primeru, najprej vsaka nit v lokalni pomnilnik prenese po en podatek iz prve in druge matrike, množenje pa poteka podobno kot pri osnovnem algoritmu. Edina razlika je, da niti dostopajo do lokalnega pomnilnika, do globalnega pomnilnika pa samo, kadar morajo prenesti novo ploščico. Meritve časa izvajanja v tabeli 5.12 zopet kažejo, da z večjimi skupinami samo izboljšujemo rezultate.

	Čas [s]	
Velikost skupine	AB	CD
4 x 4	$2,94 \pm 0,04$	$101,10 \pm 0,09$
8 x 8	$0,94 \pm 0,04$	$27,98 \pm 0,04$
16 x 16	$0,62 \pm 0,11$	$16,62 \pm 0,07$
32 x 32	$0,57 \pm 0,13$	$13,14 \pm 0,01$

Tabela 5.12: Rezultati meritev za sistem 3 - uporaba lok. pomn.

Tesla K20 je skalarne narave, zato uporaba vektorskih tipov načeloma ne prinaša pohitritev, saj bi se v našem primeru ščepec prevedel tako, da bi množenja izvedel serijsko. Kljub temu pa pozna širše ukaze tipa naloži/shrani, zato bi bilo dobro ugotoviti, ali bi z uporabo teh ukazov lahko povečali prepustnost globalnega pomnilnika in s tem pohitrili izvajanje. Za namene testiranja bomo uporabili tip $float4$, kar pomeni, da moramo velikost globalne in lokalne dimenzije 0 zmanjšati za 4-krat. Dodaten pogoj je, da je velikost skupine W deljiva s 4, da lahko ohranimo ploščice kvadratne. Ker je produkt AB premajhen in so posledično napake meritev prevelike, smo algoritem testirali samo na produktu CD.

Podlago za množenje dobimo pri tretji ideji iz sistema 1. Nit (i, j) prenese vektor iz vrstice i prve ploščice, nato pa posamezno komponento iz tega

vektorja pomnoži z istoležnim vektorjem v stolpcu j druge ploščice, vmesne rezultate pa shranjuje v akumulator. Postopek se ponavlja, dokler skupina ne opravi z vsemi ploščicami. OpenCL nam z uporabo vektorskih tipov prihrani precej dela, saj sam prevede kodo tako, da upošteva dejstvo, da je Tesla K20 skalarna enota. Rezultati meritev v tabeli 5.13 kažejo, da so bile naše ugotovitve pravilne. Čas izvajanja za produkt CD smo s skupino velikosti 8x32 uspeli zmanjšati na 6,83 sekund. Za skupino velikosti 16x32 so meritve pokazale manjše poslabšanje, kar je verjetno posledica prevelike porabe lokalnega pomnilnika in registrov. V tem primeru se zmanjša število skupin, ki se lahko sočasno izvajajo.

	Čas [s]
Velikost skupine	CD
1 x 4	$121,74 \pm 0,06$
2 x 8	$27,92 \pm 0,06$
4 x 16	$9,86 \pm 0,02$
8 x 32	$6,83 \pm 0,04$
16 x 32	$8,94 \pm 0,09$

Tabela 5.13: Rezultati meritev za sistem 3 - uporaba lok. pomn. in vektorskih tipov

5.4.4 Sklep

Klasično množenje matrik je precej paralelen algoritem, dobro lahko izkoristimo tako vektorsko enoto kot večjedernost sistemov. Zaradi velike soodvisnosti igra uporaba predpomnilnikov na CPE in lokalnega pomnilnika na GPE zelo veliko vlogo. Najbolje se je izkazal na GPE, kjer smo z uporabo lokalnega pomnilnika, velikega števila niti na skupino in vektorskih ukazov tipa naloži/shrani, dobili precejšne pohitritve. Dodatna ugotovitev je, da z uporabo vektorske enote na sistemih, ki podpirajo vektorske operacije, nadomestimo niti, kar se kaže v tem, da večje število niti ne igra ključne vloge pri učinkovitosti. Iz meritev lahko opazimo, da se Xeon Phi izkaže slabše od Tesle K20. Klasična sočasna večnitnost očitno ne zmore tako dobro izkori-

stiti virov in prikriti latence, kot veliko število lahkih niti na Tesla K20 in to navkljub velikemu številu računskih enot in široke vektorske enote.

Pri vektorskih enotah smo predpostavili, da je velikost dimenzije deljiva s širino vektorske enote. Če bi želeli poljubno velikost, bi lahko dimenzijo matrike podaljšali, tako, da bo deljiva s širino vektorske enote, pri čemer, bi odvečni del napolnili z ničlami. V tem primeru imamo malo odvečnega prenosa podatkov in dela z množenjem, rezultat pa bi bil še vedno pravilen.

5.5 Predponska vsota

V tem delu bomo prilagodili predponsko vsoto za vse dane sisteme. Pri tem predpostavljamo, da je velikost problema enaka za vse tri sisteme, in sicer 10^8 elementov.

5.5.1 Rešitev za sistem 1

CPE ne poznajo eksplicitnega predpomnilnika, vse v zvezi s prenašanjem podatkov v predpomnilnike je rešeno na strojnem nivoju. Posledično uporaba lokalnega pomnilnika nima nobenega smisla, saj ne bi povečala števila zadetkov v predpomnilniku. Pravzaprav bi uporaba lokalnega pomnilnika zgolj povečala časovno režijo zaradi odvečnega kopiranja na začetku in prenašanja obdelanih podatkov nazaj. Da niti v zadnji skupini ne bi posegale izven tabele, je treba tabelo podaljšati za toliko, da bo deljiva z velikostjo lokalnega problema, to je dvakratnik števila niti na skupino. CPE ponuja še uporabo vektorske enote. Izkoristimo jo lahko v operaciji prištevanja, ko vsaka skupina svojim elementom prišteje vsoto predhodne skupine. Vektorska enota omogoča, da seštevamo do 4 elemente naenkrat, zato globalno in lokalno velikost skupine zmanjšamo za štirikrat.

Testne meritve v tabeli 5.14 kažejo, da od določene točke dalje z večanjem skupine zgolj poslabšujemo učinkovitost. Razlog je najverjetneje v tem, da zaradi večje skupine povečujemo število zgrešitev v predpomnilniku, saj z večjimi koraki poslabšujemo učinkovitost predpomnilnikov, poleg tega pa

tudi precej povečamo komunikacijo med nitmi zaradi uporabljenih preprek. Idealna velikost je 16 do 32 niti na skupino.

Število niti	Čas [s]
4	$2,24 \pm 0,03$
8	$2,00 \pm 0,02$
16	$1,98 \pm 0,01$
32	$2,04 \pm 0,01$
64	$2,14 \pm 0,00$
128	$2,31 \pm 0,03$
256	$2,64 \pm 0,01$
512	$2,83 \pm 0,00$
1024	$3,06 \pm 0,02$

Tabela 5.14: Rezultati meritev za sistem 1 - predponska vsota

5.5.2 Rešitev za sistem 2

Uporabimo lahko rešitev iz sistema 1 s to razliko, da lahko izkoristimo večjo vektorsko enoto, s katero seštejemo do 16 elementov na enkrat. Testne meritve v tabeli 5.15 kažejo, da je optimalna velikost skupine 64 niti na skupino. Z nadaljnim večanjem skupine podobno kot pri sistemu 1 nekoliko malo poslabšujemo učinkovitost zaradi večje komunikacije niti zaradi uporabe preprek.

Število niti	Čas [s]
16	$2,68 \pm 0,11$
32	$2,58 \pm 0,09$
64	$2,39 \pm 0,02$
128	$2,45 \pm 0,17$
256	$2,48 \pm 0,13$
512	$2,57 \pm 0,14$
1024	$2,56 \pm 0,16$
2048	$2,57 \pm 0,10$

Tabela 5.15: Rezultati meritev za sistem 2 - predponska vsota

5.5.3 Rešitev za sistem 3

GPE zaradi visoke latence združujejo dostope do globalnega pomnilnika po snopih niti in s tem poskušajo čim bolj minimizirati število dostopov. To je možno, kadar so dostopi niti poravnani. Kadar pa se dostopi opravljajo z določenim korakom oziroma so bolj naključni, začne učinkovitost precej hitro padati. Rešitev za ta problem je lokalni pomnilnik, ki omogoča hiter dostop na nivoju posamezne niti. Vsaka skupina najprej v lokalni pomnilnik prenese del tabele iz globalnega pomnilnika. Ker se skupine na GPE izvajajo po snopih 32 niti, vsaka nit najprej prenese element iz prve polovice tabele in nato iz druge. Tako zmanjšamo število dostopov do pomnilnika, saj so prenosi niti združeni. Težavo z zadnjo skupino, kjer bi niti lahko dostopale izven tabele, rešimo tako, da v lokalni pomnilnik preprosto prenesemo vrednost 0. Vsaka skupina nato izvede algoritem, opisan v poglavju 4.3, nad tabelo v lokalnem pomnilniku in na koncu na enak način shrani rezultate v globalni pomnilnik. Ne smemo pozabiti na uporabo preprek, da se vsi rezultati niti vpišejo v lokalni pomnilnik po vsakem prenašanju in pred vsako iteracijo, sicer lahko pride do prepisovanja rezultatov.

Z uporabo lokalnega pomnilnika smo naleteli na novo težavo, ki nastane zaradi vzorca, kako niti dostopajo do lokalnega pomnilnika. Lokalni pomnilnik je namreč razdeljen na banke. Težava nastane, ko več niti v istem snopu dostopa do iste banke, pri čemer niti dostopajo do različnih naslovov. V tem primeru pride do konflikta v dostopu do banke in posledično se morajo dostopi izvesti eden za drugim. V primeru predponske vsote uporaba navideznega binarnega drevesa pomeni, da se razkorak niti v dostopu do pomnilnika podvoji na vsakem višjem nivoju in pri tem tudi podvoji število niti, ki dostopajo do iste banke. Stopnja konflikta je največja na srednjem nivoju, nato pa stopnja konflikta začne upadati. Da bi se izognili vsaj večini konfliktov, mora nit i, ki dostopa do elementa z indeksom n, indeksu prišteti določen odmik (ang. padding), v tem primeru vrednost indeksa n deljeno (celoštevilsko) s številom bank, torej 32. Posledično se za toliko poveča tudi skupna poraba lokalnega pomnilnika.

Meritve kažejo, da se rezultati ne razlikujejo veliko. Zaradi več preprek je bolje uporabiti nekoliko manjšo skupino, da zmanjšamo sinhronizacijo med nitmi. Rezultati meritev v tabeli 5.16 kažejo, da z upoštevanjem bank v lokalnem pomnilniku nismo pridobili veliko. Da ne bi prišli do napačnih sklepov smo meritve ponovili še na večjem problemu, to je 10^9 elementov. Meritve v tabeli 5.17 zopet kažejo le malenkostno izboljšanje in to predvsem pri večjih skupinah. Razlog za to je, da imamo pri večjih skupinah tudi več konfliktov, višje pa so tudi stopnje konfliktov. Optimalna velikost skupine je od 128 do 256 niti na skupino, pri večjih pa začne učinkovitost zopet padati zaradi večje komunikacije znotraj skupine.

	Čas [s]	
Število niti	Predponska vsota	Predponska vsota brez konfliktov
32	$0,89 \pm 0,09$	$0,83 \pm 0,04$
64	$0,88 \pm 0,09$	$0,84 \pm 0,07$
128	$0,87 \pm 0,10$	$0,87 \pm 0,05$
256	$0,82 \pm 0,04$	$0,78 \pm 0,01$
512	$0,84 \pm 0,04$	$0,83 \pm 0,04$
1024	$0,84 \pm 0,04$	$0,84 \pm 0,05$

Tabela 5.16: Rezultati meritev za sistem 3 - predponska vsota - $N = 10^8$

	Čas [s]	
Število niti	Predponska vsota	Predponska vsota brez konfliktov
32	$5,54 \pm 0,02$	$5,52 \pm 0,03$
64	$5,37 \pm 0,04$	$5,28 \pm 0,02$
128	$5,20 \pm 0,04$	$5,15 \pm 0,04$
256	$5,32 \pm 0,05$	$5,18 \pm 0,03$
512	$5,42 \pm 0,02$	$5,24 \pm 0,03$
1024	$5,59 \pm 0,04$	$5,32 \pm 0,03$

Tabela 5.17: Rezultati meritev za sistem 3 - predponska vsota - $N = 10^9$

5.5.4 Sklep

Ugotovili smo, da je v primeru več preprek zaradi sinhronizacije med nitmi bolje uporabiti manjše skupine. Na GPE smo z upoštevanjem bank lokalnega pomnilnika spremenili algoritem tako, da smo precej zmanjšali število konfliktov v dostopu do iste banke. Rezultati sicer niso prikazali bistvenih izboljšav, bi pa se verjetno bolje izkazalo na kakšni starejši GPE.

5.6 Problem n teles

Za vse tri arhitekture bomo prilagodili problem n teles. Merili bomo čas izvajanja pri različnem številu niti za 10000 teles in 1000 ponovitev.

5.6.1 Rešitev za sistem 1

Posamezna nit naprej prenese podatke o izbranem telesu v zasebni pomnilnik, nato pa na podlagi vseh ostalih teles izračuna prispevke sil na telo ter na podlagi tega še novo hitrost in pozicijo. Vektorizacijo v celoti prepuščamo prevajalniku, saj jo je težko pametno izkoristiti.

Poizvedba ščepca nam pokaže, da lahko na skupino uporabimo največ 64 niti, čeprav je fizična omejitev 1024. To je posledica večje porabe virov, predvsem zasebnega pomnilnika, saj ima ščepec precej vhodnih argumentov pa tudi precej spremenljivk. Na CPU zasebni pomnilnik fizično predstavlja kombinacija sklada procesa in nekaj registrov. Ker je sklad procesa omejen, je omejeno tudi število niti na skupino. Rezultati meritev v tabeli 5.18 kažejo najboljšo učinkovitost izvajanja pri 16 nitih na skupino, pri večjih skupinah pa zaradi omejitev zasebnega pomnilnika učinkovitost pade.

5.6.2 Rešitev za sistem 2

Rešitev je enaka kot pri sistemu 1. Meritve v tabeli 5.19 kažejo najboljšo učinkovitost nekje pri 4 do 32 niti na skupino. Pri večjih skupinah začne čas izvajanja naraščati precej hitro. Razlog je v tem, da imamo relativno

Število niti	Čas [s]
1	$770,09 \pm 0,01$
2	$769,96 \pm 0,11$
4	$770,52 \pm 0,44$
8	$652,38 \pm 0,29$
16	$639,14 \pm 0,48$
32	$655,78 \pm 2,93$
64	$656,64 \pm 6,25$

Tabela 5.18: Rezultati meritev za sistem 1 - problem n teles

majhno globalno število niti. Večja skupina pomeni manjše število skupin, ki se lahko sočasno izvajajo po računskih enotah. Xeon Phi ima 236 računskih enot, kar predstavlja tudi minimalno število skupin, za dobro izkoriščenost pa se priporoča vsaj 1000 skupin. To, se kaže tudi v našem primeru, ko zaradi prevelikih skupin začne padati učinkovitost, saj Xeon ne more dovolj dobro prikrivati latence s preklapljanjem med skupinami, pa tudi sama izkoriščenost računskih enot se poslabša, saj nekatere enote ne počnejo ničesar.

Število niti	Čas [s]
1	$197,00 \pm 0,24$
2	$27,18 \pm 0,17$
4	$27,19 \pm 0,07$
8	$27,32 \pm 0,12$
16	$27,24 \pm 0,09$
32	$31,80 \pm 0,13$
64	$36,00 \pm 0,14$
128	$55,52 \pm 0,04$
256	$85,71 \pm 0,01$
512	$169,57 \pm 0,03$

Tabela 5.19: Rezultati meritev za sistem 2 - problem n teles

5.6.3 Rešitev za sistem 3

Sila na vsako telo je enaka vsoti prispevkov vseh ostalih teles na izbrano telo. To pomeni, da imamo veliko ponovne uporabe istih podatkov, saj je vsako

telo odvisno od vseh drugih teles. Za učinkovito izvajanje je torej uporaba lokalnega pomnilnika nuja.

Vsaka nit v svoj zasebni pomnilnik prenese podatke o telesu, glede na svoj globalni indeks, saj želimo, kolikor je mogoče zmanjšati število dostopov do globalnega pomnilnika. Celotna skupina v lokalni pomnilnik prenese podatke o drugih telesih. Algoritem nato izračuna silo in pospeške telesa, pri čemer dostopa do lokalnega pomnilnika. Ker je lokalni pomnilnik omejen, se mora postopek ponoviti tolikokrat, kolikor je skupin, zato, da obravnavamo vsa telesa.

Prednost algoritma je, da vse niti v skupini dostopajo do iste pozicije v lokalnem pomnilniku. To pomeni, da lahko lokalni pomnilnik uporabi raztros, kjer se vsem nitim v snopu pošlje isti podatek. Tako lahko popolnoma izniči konflikte v dostopu do iste banke, saj bi v nasprotnem primeru imeli konflikt stopnje 32. To pomeni, da bi vseh 32 niti v snopu dostopalo do iste banke, zato bi se morali dostopi opraviti zaporedoma za vsako nit posebej, kar pa bi upočasnilo izvajanje.

Meritve časov izvajanja so navedene v tabeli 5.20 in kažejo najboljšo učinkovitost pri uporabi 128 in 256 niti na skupino. Pri večjih skupinah začne učinkovitost padati, saj zaradi manjšega števila skupin nekatere računske enote postanejo neizkoriščene, dodatno težavo pa predstavlja tudi večja poraba lokalnega pomnilnika in registrov, zaradi česar se lahko sočasno na isti računski enoti izvaja manjše število skupin. Pri 1024 nitih na skupino opazimo malenkostno izboljšanje. Razlog je, da kljub manjšemu številu skupin, Tesla K20 bolje izkoristi računske enote na nivoju niti in ne toliko na sočasnem izvajanju več skupin na računski enoti.

5.6.4 Sklep

Problem n teles je predvsem računsko zahteven problem. Vektorske enote ne moremo izkoristiti, pa tudi, če bi jo, ne bi prinesla boljše učinkovitosti, ker bi zgolj povečali delo posamezni niti, ne pa tudi izkoristili vektorskih enot. Zato se je v tem primeru bolje zanesti na implicitno vektorizacijo. Velikost

Število niti	Čas [s]
32	$47,71 \pm 0,05$
64	$28,20 \pm 0,07$
128	$26,29 \pm 0,10$
256	$25,28 \pm 0,11$
512	$29,93 \pm 0,09$
1024	$28,56 \pm 0,05$

Tabela 5.20: Rezultati meritev za sistem 3 - problem n teles

skupine določa število skupin, to pa ima lahko velik vpliv na učinkovitost izvajanja. Vsaka skupina se namreč izvaja na eni računski enoti in če je število skupin premajhno, začne učinkovitost izvajanja padati, še posebej, če je število računskih enot veliko. Če je velikost problema sorazmerno majhna, je pametno glede na število računskih enot zmanjšati velikost skupine. Kadar je skupina premajhna, prevlada cena režije preklopa med skupinami, zato učinkovitost izvajanja upade.

5.7 Bitonično urejanje

V tem poglavju bomo za vse tri arhitekture prilagodili bitonično urejanje in merili čas izvajanja pri različnem številu niti za 2^{26} elementov.

5.7.1 Rešitev za sistem 1

Za boljšo učinkovitost moramo izkoristiti vektorsko enoto, vendar smo na žalost precej omejeni. Iz slike 4.3, lahko opazimo, da velikost zaporedja vsako fazo eksponentno narašča. Vsaka faza je nadalje sestavljena iz iteracij, kjer pa velikost zaporedja eksponentno pada do zaporedja velikosti dve, ko se celotna faza konča. Ideja je, da sedaj uporabimo dva vektorja štirih števil, torej primerjamo po osem števil na enkrat. Globalna dimenzija problema se tako zmanjša za 4-krat, pri čemer pa še vedno uporabljamo isti princip kot pri običajnem urejanju, le da tu primerjamo dva vektorja na enkrat namesto posameznih števil. To lahko naredimo samo v iteracijah, kjer je velikost

zaporedja vsaj osem. Če je manjša, uporabimo običajen algoritem. Glede na rezultat primerjave nato zamenjamo elemente med vektorjema, na koncu pa ju še shranimo nazaj v pomnilnik. Prednost vektorizacije je, da elementne menjamo med registri, zmanjšamo globalno število niti in s tem nepotrebno režijo, saj polovica niti ne počne ničesar ter omogoča uporabo širših ukazov naloži/shrani, kar poveča prepustnost. Uporabljamo torej dva ščepca. Eden sortira po običajnem algoritmu za iteracije, kjer je velikost zaporedja manjša od osem, drugi pa uporablja vektorsko enoto za večje velikosti zaporedja.

Vektorska enota poveča količino dela na posamezno nit in zmanjša število skupin, kar je dobro, saj imamo malo računskih enot, obenem pa omogoča večjo pomnilniško prepustnost. To je tudi razlog, da se večje skupine bolje obnesejo od manjših, kar kažejo rezultati meritev v tabeli 5.21.

Število niti	Čas [s]
1	$126,25 \pm 2,72$
2	$74,78 \pm 0,22$
4	$49,15 \pm 0,13$
8	$37,41 \pm 0,24$
16	$31,25 \pm 0,03$
32	$28,33 \pm 0,03$
64	$26,68 \pm 0,04$
128	$25,74 \pm 0,02$
256	$25,31 \pm 0,01$
512	$25,18 \pm 0,17$
1024	$25,08 \pm 0,18$

Tabela 5.21: Rezultati meritev za sistem 1 - bitonično urejanje

5.7.2 Rešitev za sistem 2

Ideja je podobna kot pri sistemu 1, s to razliko, da ima Xeon Phi 16-kratno vektorsko enoto, s čimer lahko opravimo do 16 primerjav na enkrat. Program iz sistema 1 nadgradimo tako, da poleg 4-kratne vektorske enote izkoristimo še 8-kratno, za zaporedja velikosti 16 in 16-kratno vektorsko enoto za zaporedja velikosti 32 ali več. Na ta način precej zmanjšamo globalno število

vseh niti in se znebimo nepotrebne režije, kjer vsako iteracijo polovica niti ne naredi ničesar. Rezultati meritev v tabeli 5.22 kažejo najboljšo učinkovitost nekje pri 128 do 256 nitih na skupino. Večje skupine ne prinašajo niti večjih izboljšav niti večjih poslabšanj. Zaradi vektorizacije smo namreč že precej dobro izkoristili posamezna jedra, režija zaradi preklopa skupin pa izgubi na pomenu.

Število niti	Čas [s]
1	$17,22 \pm 0,10$
2	$6,95 \pm 0,11$
4	$5,84 \pm 0,11$
8	$5,36 \pm 0,11$
16	$4,98 \pm 0,07$
32	$4,49 \pm 0,05$
64	$4,29 \pm 0,03$
128	$4,09 \pm 0,04$
256	$4,16 \pm 0,13$
512	$4,23 \pm 0,15$
1024	$4,10 \pm 0,05$
2048	$4,14 \pm 0,05$

Tabela 5.22: Rezultati meritev za sistem 2 - bitonično urejanje

5.7.3 Rešitev za sistem 3

GPE za dobro učinkovitost zahtevajo uporabo lokalnega pomnilnika. Težava lokalnega pomnilnika je, da je precej majhen, zato lahko v lokalnem pomnilniku izvedemo le prvih nekaj faz, odvisno od velikosti skupine. Nadalje vsaka faza dela neposredno z globalnim pomnilnikom tako kot pri običajnem algoritmu, dokler se velikost zaporedja ne zmanjša ponovno na velikost lokalne skupine. V tem primeru lahko celotno zaporedje prenesemo v lokalni pomnilnik, kjer opravimo operacijo zlivanja in na koncu zaporedje prenesemo nazaj. Uporabljamo tri ščepce. Prvi ureja elemente lokalno, in sicer prvih nekaj faz, dokler ni velikost zaporedja večja od velikosti skupine. Drugi izvaja zlivanje nad globalnim pomnilnikom, dokler je velikost zaporedja večja od velikosti skupine, nato pa preklopi na ščepec, ki celotno zlivanje do konca opravi v lokalnem pomnilniku. Postopek na sliki 4.3 v poglavju 4, bi v primeru velikosti skupine 4, celotne prve dve fazi opravil v lokalnem pomnilniku. Prvi del tretje faze bi opravil v globalnem pomnilniku, nato pa spet lokalnem. V četrti fazi bi prvo polovico opravil v globalnem pomnilniku, drugo pa v lokalnem.

Število niti	Čas [s]
32	$3,23 \pm 0,05$
64	$1,96 \pm 0,06$
128	$1,48 \pm 0,08$
256	$1,43 \pm 0,03$
512	$1,47 \pm 0,03$
1024	$1,57 \pm 0,04$

Tabela 5.23: Rezultati meritev za sistem 3 - bitonično urejanje

Glede na izvajanje ščepca domnevamo, da bi se splačalo uporabiti čim večjo delovno skupino, saj lahko tako več dela opravimo v lokalnem pomnilniku in tako zmanjšamo režijo zaradi zagonov novih ščepcev. Rezultati meritev v tabeli 5.23 kažejo, da učinkovitost z večanjem delovne skupine narašča, vendar najboljšo učinkovitost dosežemo pri 256 do 512 niti na skupino. Nadalje se zaradi večje porabe lokalnega pomnilnika in registrov zmanjša število

skupinih, ki se lahko sočasno izvajajo, kar ima očitno večjo režijo, kot zagon novega ščepca. Učinkovitost izvajanja je posledično manjša.

5.7.4 Sklep

Bitonično urejanje je izredno paralelen algoritem za urejanje, saj najbolj učinkovito deluje, na sistemih z veliko jedri. Bitonično urejanje se najbolj učinkovito izvaja na GPE, saj lahko z uporabo lokalnega pomnilnika posamezne dele faze uredimo v lokalnem pomnilniku, ki nima težav z visoko latenco in prihrani režijo pri zagonu novih ščepcev. Algoritem tudi dobro izkoristi pasovno širino in vektorsko enoto, saj dobra zasnova algoritma dobro izkorišča princip lokalnosti.

5.8 Ugotovitve iz testnih primerov

OpenCL deluje razmeroma dobro na različnih sistemih, vendar to ni dovolj, kadar želimo, da se program na dani arhitekturi izvaja kar se da učinkovito. Težava nastane zaradi arhitekturnih razlik med sistemi, zato moramo programe ustrezno prilagoditi. Prilagoditve obsegajo število niti na skupino, količino dela na nit, uporabo vektorske enote in lokalnega pomnilnika ter prikrivanje pomnilniške latence. Namen tega poglavja je podati smernice, ki jih moramo upoštevati pri prilagajanju programov za čim bolj učinkovito izvajanje, pri čemer upoštevamo tako spoznanja iz teorije kot tudi meritev iz prejšnjih primerov. Pri opisovanju se opiramo na OpenCL poizvedbe, ki so bolj podrobno opisane v poglavju 3.6.

Za posamezno skupino je značilno, da se lahko izvaja na eni računski enoti, medtem ko se na eni računski enoti lahko sočasno izvaja več skupin. Iz tega sledi, da moramo vedno imeti več skupin, kot je računskih enot, zato da lahko dovolj dobro izkoristimo vse računske enote. Hhrati mora biti posamezna skupina dovolj velika, da zmanjšamo režijo pri preklopu med skupinami in dobro prikrijemo pomnilniško latenco. Slednja predstavlja težavo predvsem na GPE, zato morajo biti skupine dovolj velike, da lahko enostavno

preklopijo med snopi niti, poleg tega pa je menjava konteksta niti na GPE praktično zastonj.

Pri določanju velikosti skupine si lahko pomagamo s poizvedbo za priporočen večkratnik števila niti. To je še posebej pomembno na napravah kot so GPE, ki ščepec izvajajo po snopih, saj lahko le tako dobro izkoristimo vse procesne enote naprave. Podobno naj bo na vektorskih napravah število niti večkratnik širine vektorske enote, saj tako prevajalniku omogočimo implicitno vektorizacijo. Kadar je število računskih enot majhno, je včasih dobro, če posamezni niti dodelimo več dela in s tem navidezno povečamo velikost skupine. To smo lahko opazili v primeru histograma na CPE, kjer smo na ta način precej zmanjšali ceno redukcije skupin v globalni pomnilnik. Dodatno za CPE velja, da imajo majhno število računskih enot z veliko dodatne logike za hitrejše sekvenčno izvajanje, kot je recimo predikcija skokov. Zato včasih velja premisliti, da bi povečali količino dela na nit z uporabo zank.

Kadar povečujemo velikost skupine, moramo vedeti, da s tem povečujemo porabo virov kot so zasebni in lokalni pomnilnik ter procesni elementi. Iz meritev namreč opazimo, da se v najboljšem primeru učinkovitost izvajanja zaradi polne zasedenosti enot ne izboljšuje več, nasprotno pa v najslabšem pa povzroči zmanjšanje števila skupin, ki se sočasno izvajajo na računski enoti in posledično učinkovitost izvajanja pade. Velike skupine niso priporočljive tudi v primeru, kadar ščepec vsebuje prepreke, saj to poveča komunikacijo med nitmi zaradi sinhronizacije. Še ena stvar je kompleksnost samega ščepca. Tipično se večje skupine bolje izkažejo, kadar je ščepec relativno enostaven in ne porabi veliko virov, ali pa če je algoritem bolj pomnilniško zahteven in ima malo računskih operacij, zato ne more pokriti latence drugače kot s preklopom niti. Optimalno velikost skupine je torej težko določiti, zato si lahko v praksi pomagamo s poskusi.

Na vektorskih napravah moramo za dobro učinkovitost izkoristiti vektorsko enoto. Do določene mere je pri tem uspešen že prevajalnik, ki ščepec vektorizira tako, da združuje niti, vendar to tipično ni dovolj za optimalno izvajanje. Če želimo popolnoma izkoristiti vektorske enote, moramo ščepec

vektorizirati ročno z uporabo vektorskih tipov in vektorskih operacij. Vektorski tipi v OpenCL so prenosljivi na nivoju izvorne kode, vendar je, kadar ščepec vektoriziramo ročno, pametno upoštevati širino vektorske enote naprave za določen podatkovni tip. To najlažje izvemo s pomočjo poizvedb OpenCL. Z uporabo pravega vektorskega tipa dosežemo najboljšo izkoriščenost računske enote na račun tega, da žrtvujemo prenosljivost programa na nivoju učinkovitosti izvajanja. Kljub temu lahko včasih uporaba večjih vektorskih tipov dodatno poveča učinkovitost, ker povečuje zadetke v predpomnilniku. S pravilno ročno vektorizacijo popolnoma izkoristimo računske enote, zato velikost skupine izgubi večji pomen.

Na skalarnih enotah tipično nima smisla uporabiti vektorskih tipov, saj se bo ščepec prevedel tako, da bo upošteval to dejstvo. Kljub temu je včasih to koristno, saj na enostaven način povečamo količino dela na nit, poleg tega pa lahko povečamo pomnilniško prepustnost zaradi širših ukazov tipa naloži/shrani.

Del abstraktnega pomnilniškega modela OpenCL predstavlja lokalni pomnilnik, ki je v svoji osnovi namenjen za komunikacijo niti znotraj skupine. Vsaka računska enota ima svoj lastni lokalni pomnilnik, razdeljen med več skupinami. Iz tega sledi, da velikost uporabljenega lokalnega pomnilnika vpliva na število skupin, ki se sočasno izvajajo na računski enoti. Velikost uporabljenega lokalnega pomnilnika na skupino naj bo premo sorazmerna številu niti v skupini. Na ta način zagotovimo, da nobena nit ne čaka, če pa so skupine majhne, se lahko na računski enoti še vedno sočasno izvaja več skupin.

Običajno se lokalni pomnilnik uporablja le na arhitekturah, ki imajo podporo za eksplicitno upravljanje s predpomnilnikom, kot so GPE in služi skrivanju visoke pomnilniške latence. V nasprotnem primeru uporaba lokalnega pomnilnika nima večjega smisla, ker predstavlja zgolj dodatno časovno režijo. V teh primerih je edina rešitev uporaba čim bolj zaporednih dostopov, da čim bolje izkoristimo predpomnilnike. Izjemoma bi uporaba lokalnega pomnilnika koristila, da zmanjšamo komunikacijo med predpomnilniki in vpliv

atomičnih operacij, ali pa kjer bi povečali zadetke v predpomnilniku, kot je procesiranje podatkov po stolpcu tabele, kjer imamo velike korake med sosednjimi elementi.

Splošna primerjava algoritmov med CPE Intel Core i5 in mnogojedrnikom Intel Xeon Phi kaže, da so si precej podobni. Tudi arhitekturno imata kar nekaj podobnosti, kot je strojno predpomnenje in uporaba vektorskih enot. Posamezna jedra so pri obeh kompleksna, saj vsebujejo na kupe dodatne logike za hitrejše sekvenčno delovanje in podporo operacijskim sistemom. Razlika je v frekvenci in številu jeder, zato moramo število skupin prilagoditi tako, da bo izkoriščenost polna. Specifično moramo prilagoditi tudi velikost vektorskih tipov tako, da se ujema s širino vektorske enote. Nvidia Tesla K20 GPE ima veliko število enostavnih jeder skalarne narave, ki delujejo z nizko frekvenco in so narejena za sočasno izvajanje velikega števila niti. Da prikrijemo visoko latenco, moramo namesto uporabe vektorskih tipov izkoristiti lokalni pomnilnik in veliko število niti, s katerim prekrivajo visoko latenco globalnega pomnilnika.

Poglavje 6

Sklepne ugotovitve

Skozi diplomsko delo smo skušali ugotoviti kaj vse moramo upoštevati pri pisanju programov OpenCL, da se bodo učinkovito izvajali na danem sistemu. Pri učinkovitosti imamo v mislih predvsem, kako izkoristiti paralelizem tako na nivoju ukazov kot na nivoju večnitnosti. Glavna težava, ki se tu pojavi, so arhitekturne razlike med sistemi, ki onemogočajo, da bi program enako učinkovito tekel na vseh sistemih. Zato je edina možnost, da programe prilagodimo za vsak sistem posebej. Prilagoditve programov obsegajo upoštevanje števila računskih enot, število niti na skupino, prisotnost in širina vektorske enote, uporabo lokalnega pomnilnika in upoštevanje predpomnilnikov, seveda pa bi lahko marsikaj avtomatizirali z upoštevanjem lastnosti sistema. V tej diplomski nalogi smo pet programov, to so histogram, množenje matrik, predponska vsota, problem n teles in bitonično urejanje, prilagodili trem različnim sistemom tako, da se izvajajo čim bolj učinkovito. Prvi sistem je CPE Intel Core i5-2450M, drugi je mnogojedrnik Xeon Phi 5110P, tretji pa GPE Nvidia Tesla K20. Pri tem smo za različne možne izboljšave in pri različnem številu niti na skupino merili čas izvajanja teh programov.

Čeprav je težko točno povedati, kdaj se bo posamezen algoritem najbolj učinkovito izvajal na nekem sistemu, je v prvi vrsti pomembna dobra izkoriščenost računskih in procesnih enot. To pomeni, da moramo imeti dovolj

skupin, da se program učinkovito razporedi po računskih enotah in obenem dovolj niti, da zmanjšamo režijo zaradi preklopa skupin. Vendar pa z večanjem skupine večamo tudi porabo virov, zato lahko učinkovitost izvajanja pade. Kjer lahko, moramo izkoristiti vektorsko enoto. Do določene mere nam pri tem pomaga prevajalnik, vedar je veliko bolj učinkovito, če program ročno vektoriziramo, saj bo edino na ta način vektorska enota najbolje prišla do izraza. Na GPE predstavlja težavo predvsem visoka latenca globalnega pomnilnika. Ta problem rešujemo z velikimi skupinami, poravnanimi dostopi do globalnega pomnilnika oziroma uporabimo lokalni pomnilnik, ki je manjši, a bistveno hitrejši.

Pri primerjavi algoritmov je opazna podobnost med CPE in mnogojernikom Xeon Phi, kar je posledica arhitekturnih podobnosti. Tako sta oba sistema vektorske narave, s predpomnilnikom pa upravlja stojna oprema. Razlike nastajajo v podrobnostih, kot je število in kompleksnost računskih enot, širini vektorske enote ipd. Tesla K20 GPE je skalarne narave in vektorsko enoto nadomešča z nitmi, zato morajo biti skupine velike. Za učinkovito izvajanje je nujno, da minimiziramo število dostopov do globalnega pomnilnika, čim bolje izkoristimo visoko prepustnost in uporabimo lokalni pomnilnik, ki nam služi kot eksplicitni predpomnilnik.

Literatura

[1] Intel OpenCL Optimization Guide. Dosegljivo: https://software.intel.com/sites/landingpage/opencl/optimization-guide/. [Dostopano 20. 3. 2016].

[2] The OpenCL Specification Version 1.2 Dosegljivo: https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf . [Dostopano 20. 3. 2016].

[3] B. Gaster, L. Howes, D. Kaeli, P. Mistry, D. Schaa. Heterogeneous Computing with OpenCL, 1st Edition, 2012

[4] An Introduction to the OpenCL Programming Model. Jonathan Tompson. Kristofer Schlachtery. NYU: Media Research Lab Dosegljivo: http://www.cs.bris.ac.uk/home/simonm/workshops/OpenCL_lecture4.pdf . [Dostopano 23. 3. 2016].

[5] Bitonic sorter. Dosegljivo: https://en.wikipedia.org/wiki/Bitonic_sorter. [Dostopano 28. 3. 2016].

[6] Programming GPUs Lecture 7 - Sorting in GPU. Juan J. Durillo Dosegljivo: http://www.dps.uibk.ac.at/~juan/Lecture_7.pdf. [Dostopano 28. 3. 2016].

[7] Parallel Prefix Sum (Scan) with CUDA. Dosegljivo: http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html. [Dostopano 19. 4. 2016].

[8] Tutorial 11 – GPU algorithms design - Parallel primitives – Scan operation. Dosegljivo: https://webcourse.cs.technion.ac.il/236370/Winter2010-2011/ho/WCFiles/Tutorial%2011%20-%20GPU%20Algorithm.pdf. [Dostopano 19. 4. 2016].

[9] Questions about global and local work size Dosegljivo: http://stackoverflow.com/questions/3957125/questions-about-global-and-local-work-size?rq=1 . [Dostopano 5. 5. 2016].

[10] Number of Compute Units corresponding to the number of work groups Dosegljivo: http://stackoverflow.com/questions/9326430/number-of-compute-units-corresponding-to-the-number-of-work-groups . [Dostopano 5. 5. 2016].

[11] OpenCL Programming Guide for the CUDA Architecture Dosegljivo: http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf. [Dostopano 5. 5. 2016].

[12] CUDA Memory and Cache Architecture Dosegljivo: http://supercomputingblog.com/cuda/cuda-memory-and-cache-architecture/ . [Dostopano 12. 5. 2016].

[13] Performance considerations for OpenCL on NVIDIA GPUs. Karthik Raghavan Ravi, NVIDIA Corp. Dosegljivo: http://on-demand.gputechconf.com/gtc/2016/presentation/s6382-karthik-ravi-Perf-considerations-for-OpenCL.pdf . [Dostopano 12. 5. 2016].

[14] An Introduction to CUDA/OpenCL and Manycore Graphics Processors. Dosegljivo: https://people.eecs.berkeley.edu/~demmel/cs267_Spr12/Lectures/CatanzaroIntroToGPUs.pdf . [Dostopano 18. 5. 2016].

[15] OpenCL Optimization. PengWang, NVIDIA Corp. Dosegljivo: `http://www.nvidia.com/content/GTC/documents/1068_GTC09.pdf` . [Dostopano 25. 5. 2016].

[16] OpenCL on NVIDIA GPUs. Timo Stich, NVIDIA Corp. Dosegljivo: `http://sa09.idav.ucdavis.edu/docs/SA09_NVIDIA_IHV_talk.pdf` . [Dostopano 25. 5. 2016].

[17] Introduction to GPGPU and CUDA Programming Dosegljivo: `https://cvw.cac.cornell.edu/gpu/default` . [Dostopano 25. 5. 2016].

[18] Heterogeneous Computing using openCL lecture 4 - Memory Issues. Sven-Bodo Scholz, Heriot-Watt University. Dosegljivo: `http://www.cs.bris.ac.uk/home/simonm/workshops/OpenCL_lecture4.pdf` . [Dostopano 6. 6. 2016].

[19] Autovectorizer Dosegljivo: `https://developer.apple.com/library/content/documentation/Performance/Conceptual/OpenCL_MacProgGuide/AutoVectorizer/AutoVectorizer.html#//apple_ref/doc/uid/TP40008312-CH7-SW1` . [Dostopano 25. 6. 2016].

[20] OpenCL Vectorising Features. Andreas Beckmann Dosegljivo: `http://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/opencl/opencl-vector.pdf?__blob=publicationFile` . [Dostopano 25. 6. 2016].

[21] SIMD<SIMT<SMT: parallelism in NVIDIA GPUs Dosegljivo: `http://yosefk.com/blog/simd-simt-smt-parallelism-in-nvidia-gpus.html`. [Dostopano 25. 6. 2016]

[22] Single instruction, multiple threads Dosegljivo: `https://en.wikipedia.org/wiki/Single_instruction,_multiple_threads`. [Dostopano 25. 6. 2016]

[23] OpenCL* Design and Programming Guide for the Intel® Xeon Phi™ Coprocessor Dosegljivo:

https://software.intel.com/en-us/articles/opencl-design-and-programming-guide-for-the-intel-xeon-phi-coprocessor . [Dostopano 1. 7. 2016].

[24] NVIDIA Tesla K20 GPU Accelerator (Kepler GK110) Up Close Dosegljivo: https://www.microway.com/hpc-tech-tips/nvidia-tesla-k20-gpu-accelerator-kepler-gk110-up-close/ . [Dostopano 1. 7. 2016].